Odpowiedzi na pytania ze stronki Deniziaka:
Uwagi Deniziaka do odpowiedzi studentów:
http://www.cyfronet.krakow.pl/~pedenizi/uwagi.html

I. Pytania na ocenę 3.0

I.1. Omówić typy danych języka JAVA.

Java nie posiada też typu Unsigned (bez znaku), czego konsekwencją jest to, że przekraczając zakres danego typu przejdziemy na zakres ujemny.

W Javie mamy również typ danych void. - **jest w tabeli w Thinking in Java str. 70**

Typy danych prostych

Typ	**Rozmiar (bity)**	**Klasa**
boolean	- (true, false)	Boolean
char	16 (Unicode)	Character
byte	8	Byte
short	16	Short
int	32	Integer
long	64	Long
float	32 (IEEE754)	Float
double	64 (IEEE754)	Double

I.2. Omówić znaczenie zmiennych i metod statycznych.

Zmienne mogą być porozrzucane po całym bloku i definiowane w miejscu, w którym są potrzebne. Pozwala to na bardziej nauralny styl kodowania oraz ułatwia zrozumienie takiego kodu.

Jeżeli zdefiniujesz zmienna klasy jako statyczna bedzie istniala tylko jedna taka zmienna na cala klase- w innym wypadku wszystkie obiekty beda posiadaly wlasna kopie tej zmiennej. Posiadanie zmiennych

publicznych nie jest dobrym pomyslem. Jednak stale publiczne czyli pola finalne nie stanowia zagrozenia

metody statyczne to metody ktore nie operuja na obiektach. Dla przykladu, metoda pow.Math jest metoda statyczna. Wyrazenie: Math.pow(x,y) oblicza x do potegi y. Nie potrzebuje obiektu klasy Math, by tego dokonac. Innymi slowy metoda ta nie posiada parametru ukrytego.

Metody statyczne mozna traktowac jako metody ktore nie posiadaja parametru this.

Metody statyczne nie operuja na obiektach, nie mozesz uzywac ich aby uzyskac dostep do zmiennych skladowych. Jednakze metody statyczne maja dostep do zmiennych statycznych swojej klasy. Np.

```
public static int pobierzNastepnyId(){
        return nastepnyId;//zwraca zmienna statyczna
}
```

aby wywolac taka metode podajemy nazwe jej klasy

```
int n=Pracownik.pobierzNastepnyId();
```

Metody statycznych uzywa sie w dwoch sytuacjach:

- Gdy dana metoda nie potrzebuje dostepu do obiektu, poniewaz wszystkie potrzebne jej parametry sa dostarczane jawnie(przyklad: Math.pow)
- Gdy metoda potrzebuje dostepu jedynie do zmiennych statycznych klasy(przyklad Pracownik.pobierzNastepnyId)

Mozna wywolac metode statyczna nawet jezeli nie posiadamy zadnych obiektow danej klasy. Z tego samego powodu metoda main jest metoda statyczna. Metoda main nie operuje na obiektach. W gruncie rzeczy, gdy program dopiero zaczyna działać, zadne obiekty jeszcze nie istnieja. Metoda main tworzy obiekty, ktorych potrzebuje program.

Statycznymi nie można oznaczyć: konstruktorów, nie zagnieżdżonych klas, interfejsów, lokalnych zmiennych, metod wewnętrznych (zagnieżdżonych) klas i zmiennych instancji.

Metody statycznie nie mogą być przeciążane w klasie potomnej, mogą być co najwyżej redefiniowane. Zasady stosowania elementów statycznych:

- Metody i pola statyczne mozna wywoływac z klasy,
- Metody statyczne moga sie odwoływac tylko do pól i metod statycznych,
- Pola statyczne sa wspólne dla wszystkich instancji danej klasy,
- Pola statyczne sa inicjowane wczesniej niz inne pola (przy pierwszym odwołaniu do klasy).

I.3. Co to jest polimorfizm? Jakie ma zastosowanie?

Polimorfizm oznacza możliwość traktowania obiektów różnych podtypów pewnego wspólnego typu w taki sam sposób.
Polimorfizm w programowaniu obiektowym oznacza wykorzystanie tego samego kodu do operowania na obiektach przynależnych różnym klasom, dziedziczącym od siebie. Zjawisko to jest zatem ściśle związane z klasami i dziedziczeniem
<JAVA podstawy. Helion>
Polimorfizmem nazywamy możliowśćć odwoływnia się przez obiekty do wielu różnych typów. Zmienne obiektowe w Javie są polimorficzne.

<ZNALEZIONE NA FORUM> dobra definicja:)
Polimorfizm:
Jeden z podstawowych składników programowania obiektowego.
Dzięki polimorfizmowi uzyskujemy kolejną metodę separacji interfejsu od implementacji, pozwalającą na polepszenie organizacji i czytelności kodu oraz pozwalającej na tworzenie rozszerzalnych programów.
Polimorfizm, czyli wielopostaciowość, pozwala nam traktować dany obiekt nie jako reprezentanta typu specjalizowanego, ale bazowego. Jest to możliwe dzięki zastosowaniu dziedziczenia opartego o przesłanianie funkcji klasy bazowej, dzięki czemu nie zmienia się typ obiektu. Pozwala to na pisanie kodu niezależnego od konkretnego typu. Polimorfizm jest możliwe dzięki koncepcjom takim jak:
Późne wiązanie (late binding) – przy wysyłaniu komunikatu do obiektu kod, który będzie wykonany nie jest zdeterminowany aż do czasu wykonania. Kompilator upewnia się, że metoda istnieje, sprawdza typy argumentów i typ zwracanej wartości, nie wie jednak, jaki kod należy wykonać.
W celu przeprowadzenia późnego wiązania Java umieszcza zamiast bezwzględnego wywołania specjalny fragment kodu obliczający adres ciała metody na podstawie informacji przechowywanej w obiekcie. Każdy obiekt może zatem zachowywać się inaczej, w zależności od wyniku działania tego małego fragmentu kodu. Gdy wysyłamy komunikat do obiektu, obiekt decyduje, co z nim zrobić.

Zastosowanie polimorfizmu w tworzeniu GUI.
Hierarchia klas może przekładać się na hierarchię typów. Możliwe jest wtedy podstawienie pod zmienną (lub atrybut funkcji) typu T obiektu typu S będącego podtypem T i dalsze używanie go jakby był typu T. Jest to możliwe dzięki temu, że podklasa posiada kompletny interfejs swojej nadklasy. W podklasie może być zdefiniowana metoda już istniejąca w nadklasie. Konstrukcja taka umożliwia wykonywanie operacji na obiektach bez informacji, z jakim właściwie obiektem mamy do czynienia. Rozpatrzmy typową aplikację GUI wyświetlającą na ekranie różne komponenty (np. przycisk, pole tekstowe czy listę rozwijaną).

Reagują one na te same zdarzenia: kliknięcie myszką, naciśnięcie klawisza, lecz każdy z nich reaguje inaczej, stosownie do tego czym jest. System obsługi zdarzeń najpierw określa, który z komponentów powinien obsłużyć zdarzenie, a następnie przekazuje mu je. Dzięki podtypowanie opartym na dziedziczeniu możemy utworzyć wspólną klasę Komponent z metodą obsluzKliknieciеMyszka(), którą będą rozszerzać wszystkie rodzaje komponentów. Pobierając aktywny obiekt, możemy wywołać tę metodę bez zastanawiania się czy dany obiekt jest przyciskiem czy polem tekstowym.

I.4. Na czym polega dziedziczenie i jakie ma zastosowanie?

Dziedziczenie polega na tworzeniu nowych klas na podstawie już istniejących.

Dziedziczenie jest jedną z fundamentalnych cech podejścia obiektowego. Pozwala kojarzyć klasy obiektów w hierarchie klas. W Javie hierarchia dziedziczenia dla klas ma postać drzewa. Jej korzeniem jest klasa Object. Realizacja dziedziczenia polega na tym, że klasa dziedzicząca dziedziczy po swojej nadklasie wszystkie jej atrybuty i metody (i nie ma znaczenia, czy te atrybuty i metody były zadeklarowane bezpośrednio w tej nadklasie, czy ona też odziedziczyła je po swojej z kolei nadklasie). Dziedziczenie odzwierciedla relację is-a (jest czymś). Oznacza to, że każdy obiekt podklasy jest także obiektem nadklasy. Na przykład hierarchia klas zbudowana z nadklasy Owoc i dwu podklas Jabłko i Gruszka jest prawidłowo zbudowana, bo każde jabłko i każda gruszka jest owocem. Definiujac klasy, najbardziej ogolne metody umieszczasz w nadklasie, a te bardziej wyspecjalizowane w podklasie.

W języku Java w przeciwieństwie do innych nie występuje dziedziczenie wielokrotne.To znaczy, że klasa potomna może rozszerzać tylko jedną klasę bazową. Projektanci Javy uznali, że mechanizm taki jest prostszy i nie wprowadza niepotrzebnego chaosu. Zamiast wielokrotnego dziedziczenia w Javie dostępny jest mechanizm interfejsów.

Pod-klasa nie ma dostępu do składowych prywatnych swojej nad-klasy.

Składowa zadeklarowana jako private nie jest dostępna poza klasą w której jest zadeklarowana, wliczając jej pod-klasy.

Wywołanie konstruktora nadklasy odbywa się poprzez polecenie super(lista-parametrow). W jednym konstruktorze nie moze istniec zarazem metoda super jak i this. Stosowany szczególnie gdy składowe pod-klasy przesłaniają składowe nad-klasy o tych samych nazwach. Kiedy metoda pod-klasy ma tą samą nazwę i typ jak metoda nad-klasy, wtedy mówimy że przesłania ją. Wersja metody dla nad-klasy zostaje ukryta.W hierarchii klas, najpierw wywołujemy konstruktory nad-klasy, potem pod-klasy. Jeżeli

super() nie jest użyte, wywołuje się domyślny bez-parametrowy konstruktor każdej nad-klasy.

Klasa deklarowana jako final nie posiada potomków, nie wolno po niej dziedziczyć.

final class A { ... }

Ta klasa jest nielegalna:

class B extends A { ... }

Metodę deklarowaną jako final w nad-klasie nie wolno przesłaniać w pod-klasie.

Modyfikatory z zad 8.

I.5. Co to są klasy abstrakcyjne i jakie mają zastosowanie?

Zwykle po to tworzymy klasy by tworzyć ich egzemplarze. Okazuje się jednak często, że definiujemy klasy, które z założenia nie będą nigdy miały swoich egzemplarzy. Takie klasy nazywamy **klasami abstrakcyjnymi.** Wbrew temu, co mogłoby się wydawać na pierwszy rzut oka, te klasy pełnią bardzo ważną rolę przy projektowaniu hierarchii klas. Pozwalają bowiem na wyabstrahowanie wspólnych cech wielu definiowanych klas i jawne wskazanie, że wszystkie dziedziczące klasy muszą te cechy posiadać.

Poniżej znajduje się lista najważniejszych cech klas abstrakcyjnych:

- mogą zawierać metody abstrakcyjne, czyli takie, które nie posiadają implementacji (ani nawet nawiasów klamrowych)
- może zawierać stałe (zmienne oznaczone jako public static final)
- mogą zawierać zwykłe metody, które niosą jakąś funkcjonalność, a klasy rozszerzające mogą ją bez problemu dziedziczyć
- klasy rozszerzające klasę abstrakcyjną muszą stworzyć implementację dla metod oznaczonych jako abstrakcyjne w klasie abstrakcyjnej
- metod abstrakcyjnych nie można oznaczać jako statyczne (nie posiadają implementacji)
- podobnie jak w przypadku interfejsów nie da się tworzyć instancji (obiektów) klas abstrakcyjnych

I.6. Omówić sposoby inicjalizacji pól w obiektach.

Kolejność inicjalizacji:
– pola statyczne
– deklaracje
– konstruktor

– instrukcje
✓ pola są zerowane, zmienne lokalne nie są

Konstruktorów używamy do inicjalizacji pól danych obiektu w momencie jego tworzenia. Jednakże dane statyczne klasy istnieją - jako jej atrybuty - również wtedy, gdy nie ma żadnego obiektu danej klasy. Aby można zainicjować zmienne statyczne w Javie zdefiniowano inicjator statycznych pól danych. Inicjalizacja statyczna następuje podczas pierwszego ładowania klasy do pamięci. Klasa może mieć dowolną liczbę inicjatorów statycznych. Inicjatory statyczne wykonywane są w kolejności ich wystąpienia w definicji klasy.

Inicjalizacja pól w obiektach podklas jest możliwa w przypadku ustawienia odpowiednich modyfikatorów dostępu dla danych pól w klasie nadrzędnej. Tymi modyfikatorami są: package (friendly), gdzie pole może być używane we wszystkich klasach w danym pakiecie, protected, gdzie dane pole będzie dostępne w klasie macierzystej, wszystkich podklasach (klasach dziedziczących z klasy zawierających pole/metodę) i w całym pakiecie oraz public, czyli dostęp dla wszystkich klas programu, niezależnie od pakietu.

I.7. Omówić zasady inicjalizacji obiektów.

W javie nowe obiekty sa tworzne dynamicznie w obszarze pamięci zwanym stertą (stosem) poprzez instrukcje *new*. Stosując to rozwiązanie nie wiemy, az do czasu ich wykonania, ilu obiektow potrzebujemy, jaki ma być czas ich zycia oraz dokładny typ. Kwestie te sa rozstrzygane w odpowiednim momencie podczas działania programu. Jeżeli potrzebujemy nowego obiektu, tworzymy go po prostu na stercie(sterta!=stos)
Java posiada udogodnienie zwane odsmiecaczem pamięci, automatycznie wykrywające, który obiekt nie jest już używany, a następnie niszczace go.

Inicjalizacja pol statycznych jest wykonywana tylko wtedy kiedy jest potrzebna.

Kolejnośc inicjalizacji jest nastepujaca: najpierw zmienne static, jeżeli jeszcze nie zostały zainicjalizowane przez poprzednie stworzenie obiektu, a następnie obiekty niestatyczne.

Kiedy kompilator natknie się na *new* alokuje odpowiednia ilość pamięci dla obiektu na stercie (stos). Przydzielony obszar jest wymazywany z automatycznym ustawieniem wszystkich typow podstawowych obiektu na ich wartości domyslnej (zero w przypadku liczb i jego odpowiednik dla typu boolean oraz char), a referencji na null.

Następnie inicjalizacja wszystkich pol okreslona jawnie w miejscu ich defnicji.
Kolejno wykonywane są ciała konstruktorów(może to powodować dosyć sporo działań).

Kolejność incjalizacji obiektu:

- pola statyczne
- deklaracje
- konstruktor
- instrukcje

Tablice:
– tworzenie tablicy jednowymiarowej typu int
int[] tab = new int[10];
– tworzenie tablicy wielowymiarowej
int[][] tab = new int[10][4];
– tworzenie i inicjalizacja zawartości tablicy
int[] tab = { 5, 3, 8, 2, 7 };
– liczba elementów tablicy
tab.length
– odwołanie do i-tego elementu tablicy
tab[i]

✓ tablica typów prostych jest zerowana
✓ tablica obiektów jest tablicą referencji pustych (null)

I.8. Omówić zasady ochrony dostępu do elementów klas i pakietów.

Modyfikatory:

- public – składnik klasy (pole lub metoda) widoczny dla wszystkich
- private – widoczny tylko wewnątrz danej klasy
- protected – widoczny w obrębie pakietu oraz w klasach potomnych z innych pakietów
- „puste” (friendly) – dostępne dla wszystkich wewnątrz tego samego pakietu.

Oprócz modyfikatorów dostępu istnieją jeszcze modyfikatory właściwości:

- dla klas:
    - abstract - definiuje klasę abstrakcyjną zawierającą chociaż 1 metodę abstrakcyjną (niezaimplementowaną); nie można tworzyć żadnych obiektów tej klasy; klasa taka jest użyteczna podczas budowania hierarchii klas (dziedziczenie): stanowi wzór do

tworzenia klas pochodnych;
    - final - uniemożliwia tworzenie klas pochodnych; stosuje się w celu uniemożliwienia klasa potomna "podszycia się" pod klasę bazową oraz czasami do oznaczenia klas, nad którymi prace zostały ukończone;
    - synchronizable - klasa z mechanizmem obsługi wątków (ang. threads);
- dla metod:
    - abstract - metody niezaimplementowane (bez kodu i zmiennych); użyteczna podczas budowania hierarchii klas (dziedziczenie): stanowi wzór dla metod klas pochodnych, które wymagają jej różnych implementacji;
    - static - metoda wspólna dla wszystkich obiektów danej klasy; należy ona do klasy, a nie do obiektu i może być wywoływana bez tworzenia obiektu danej klasy;
    - final - uniemożliwia klasom pochodnym "przesłanianie" metody;
    - synchronized - metoda blokująca dostęp do obiektu, do którego należy i odblokowująca go, gdy zakończy działanie; jeżeli dostęp do obiektu został wcześniej zablokowany, to metoda oczekuje na jego odblokowanie zanim zacznie się wykonywać; mechanizm bardzo istotny w przypadku programów wielowątkowych (ang. multitheads);
    - native - oznacza funkcję z wykorzystaniem nieprzenośnych cech danej platformy: programy napisane w Javie są niezależne od platformy sprzętowej i systemowej; jeżeli konieczne jest skorzystanie z mechanizmów specyficznych dla danej platformy, to modyfikator native powoduje, że program będzie mógł być uruchamiany tylko na tej 1 dedykowanej platformie;

I.9. Wyjaśnić pojęcie enkapsulacji, podać wady i zalety.

Enkapsulacja inaczej zwana **hermetyzacją** (kapsułkowaniem) jest to jedno z głównych założeń programowania obiektowego. Polega na ukrywaniu metod i atrybutów dla klas zewnętrznych. Dostęp do nich możliwy jest tylko z wewnątrz klasy, do której należą, z klas zaprzyjaźnionych lub z klas dziedziczących.

Można wyróżnić trzy główne powody wprowadzenia **hermetyzacji** do programowania obiektowego:

1. uodparnia tworzony model na błędy,
2. lepiej oddaje rzeczywistość,
3. umożliwia rozbicie modelu na mniejsze elementy.

Zalety stosowania **enkapsulacji**

- uodparnia tworzony model na błędy polegające np. na błędnym przypisywaniu wartości

- umożliwia rozbicie modelu na mniejsze elementy, które nie muszą i nie powinny obchodzić użytkownika

Wady:
Enkapsulacja nie posiada wad "fizycznych"(czy jak to tam nazwac), jednak w rękach nieudolnego programisty, który stara się ją na siłe stosować może przysporzyć problemów (czy coś w tym stylu), np. programista może ukryć ważne pole jakiejś klasy, tylko dlatego że w danym momencie z niego nie korzysta, co wcale nie oznacza że nie powinno być ono ogólno dostępne. Albo programista ukryje jakieś pole i nie stworzy metod, które umożliwiałyby korzystanie z tego pola.
Czesto enkapsulacja wprowadza problemy w interpretacji bledow kompilacji. Np. Nie pamietamy ze ktores pole jest prywatne a probujemy sie do niego odwolac.

I.10.Omówić różnice pomiędzy przeciążaniem a przesłanianiem.

W javie konstruktor wymusza **przciażanie** nazw metod. Ponieważ nazwa konstruktora jest zdeterminowana wcześniej przez nazwę klasy, może istnieć tylko jedna taka nazwa. Mimo iż przeciążanie jest koniecznością w przypadku konstruktorów, udogodnienie przez nie zapewniane ma charakter ogólny i może być używane w stosunku do każdej metody.
Każda metoda przeciażona musi pobierać unikatowa listę typów argumentów. Różnica w kolejności argumentów jest wystarczająca, ale wprowadza problemy z utrzymaniem kodu.

Zasady przeciążania metod:

- Lista argumentów musi być różna od tej z metody jaką chcemy przeciążyć, może różnić się ilością, typem parametrów, lub ilością i typem jednocześnie. Jeżeli natomiast nazwa metody oraz jej parametry będą takie same, jak w oryginale, będziemy mieć do czynienia z przesłonięciem metody (jeżeli sytuacja ma miejsce w sub klasie), lub błędem (jeżeli sytuacja ma miejsce w tej samej klasie w której znajduje się metoda, jaką przesłaniamy),
- Typ zwracany przez metodę przeciążoną, oraz jej specyfikator dostępu, mogą być inne niż w oryginale,
- Metody przeciążone mogą deklarować nowe wyjątki, lub poszerzać te, które są już zdeklarowane, pod warunkiem, że dziedziczą one po klasie Exception,
- Metoda może zostać przeciążona w ramach tej samej klasy, lub sub klasy,
- Typ referencji determinuje, która przeciążona metoda zostanie wywołana.

Ostatni punkt domaga się komentarza. W przypadku metod przesłanianych, o tym, która zostanie wykonana, decyduje typ obiektu, na jaki pokazuje referencja. W przypadku metod przeciążanych, odpowiednia metoda wybierana jest na podstawie typu referencji, a nie obiektu.

Przesłanianianie (czasem zwane tez przedefiniowanie) jest to dostarczenie w klasie pochodnej metody, ktora byla metodą wirtualna w klasie bazowej i ta definicja zastepuje metode z klasy bazowej. Roznice widac wtedy, gdy metode wywoła sie na rzecz obiektu wskazanego przez referencje do klasy bazowej (w tym rowniez wolane spod innych metod klasy bazowej).

Możemy przesłonić tylko metody z klasy bazowej oznaczone jako public, protected, oraz default.

Klasa deklarowana jako final nie posiada potomków, nie wolno po niej dziedziczyć, czyli nie można także przeciążyć ani przesłonić jej metod.

Przesłanianie i przeciążanie metod łączy się bezpośrednio z dziedziczeniem, tzn. operacje te wykonuje się na metodach z klasy bazowej.

I.11.Znaczenie i zastosowanie interfejsów w języku Java.

Interfejs to szkielet, albo bardziej szczegółowo wymóg jaki będą musiały spełniać klasy go implementujące. Natomiast sam interfejs jest całkowicie pozbawiony implementacji. Klasa ta pozwala samemu tworcy ustanowić nazwy metod, listy argumentów, typy zwracane - jednak bez cial metod. Jest to klasa bazowa o najwyższej możliwej abstrakcyjności. Pozwala na realizację odmiany “dziedziczenia wielobrazowego”, czyli tworzenia klas, które dadzą się rzutować w górę na więcej niż jeden typ bazowy. ;Arek

wymagania co do interfejsów w skrócie:

- Interfejs musi być utworzony przy użyciu słowa kluczowego interface
- Interfejsy mogą być wykorzystywane polimorficznie, tzn można ich używać jako typu ogólniejszego klas, które go implementują
- Interfejs może rozszerzać (extends) tylko interfejsy (nawet kilka, co w przypadku klas jest zabronione)
- Metody interfejsu nie mogą być zadeklarowane jako statyczne

Zastosowanie interfejsów:

- „wielodziedziczenie”
- polimorfizm
- deklaracje stałych

Interfejs w Javie to deklarowany za pomoc ą słowa kluczowego interface nazwany zbiór deklaracji zawierający:

- publiczne abstrakcyjne metody (bez implementacji),
- publiczne statyczne zmienne finalne (stałe) o ustalonych typach i wartościach.

Interfejs jest rozwiązaniem braku wielodziedziczenia w javie.

Interfejsy nie mają konstruktorów.

I.12.Zasady obsługi wyjątków.

Wyjątek jest więc mechanizmem kontroli przepływu służący do obsługi zdarzeń wyjątkowych w szczególności błędów.

Co to jest wyjatek?
- blad wykonania
- sytuacja wyjatkowa
- zadanie przerwania wykonywania bloku
- instrukcji

Aby przechwycić wyjątek rzucany przez pewien fragment kodu należy ten kod ująć w klauzulę try-catch-finally. Składnia poniżej:

```
try {
 {kod programu}
} catch( {deklaracja zmiennej dla obiektu wyjątku} ) {
 {kod obsługi błędu}
} finally {
 {kod wykonywany zawsze}
}
```

Wyjątek to obiekt, który opisuje sytuację wyjątkową(błędną) powstałą w kodzie programu:
● Kiedy powstaje błąd, wyjątek opisujący go jest "wyrzucany" w metodzie która ten błąd spowodowała.
● Metoda może "wyłapać" i "obsłużyć" wyjątek samodzielnie, lub przekazać go dalej.
● Błąd jest na koniec wyłapany i obsługiwany.

Konstrukcje Obsługi Wyjątków
● try – otacza część programu, którą chcemy monitorować na wypadek sygnalizacji błędów
● catch – w parze z try, wyłapuje określone wyjątki i obsługuje je w pewien sposób

● throw – sygnalizuje powstanie określonego wyjątku
● throws – określenie jakie wyjątki może dana metoda sygnalizować
● finally – kod, który musi być koniecznie wywołany przed opuszczeniem danej metody

Tworzenie:
- rozszerzanie interfejsu klasy
class MyException extends Exception { }

Specyfikacja:
- specyfikacja wyrzucanych wyjątków, których obsługą zajmie się kod wywołujący metodę
void method() throws IOException { }
✓ metody przeciążone – powinny wyrzucać takie same wyjątki
✓ metody implementujące – nie muszą wyrzucać wyjątków specyfikowanych przez metodę abstrakcyjną
✓ konstruktory – mogą dodawać nowe wyjątki

I.13.Zasady tworzenia apletów.

Aplet jest miniprogramem, który działa wyłącznie pod kontrola przeglądarki. Jest on ladowany automatycznie jako element strony sieciowej(tak samo jak obrazek). Kiedy aplet zostanie aktywowany, wykona swój program. Jest to jest dozu plus-zapewnia metode automatycznego rozpowszechniania oprogramowania klenta z serwera w momencie, kiedy użytkownik go potrzebuje, a nie wcześniej. Uzytkownik otrzymuje najnowsza wersje oprogramowania klienta bez borykania się z trudnościami ponownej instalacji. Java została zaprojektowana w taki sposób, ze programista tworzy tylko jeden program, który automatycznie będzie dzialal na wszystkich komputerach wyposażonych w przgladarki z wbudowanym interpreterem javy. Ponieważ java jest w pełni rozwiniętym jezykiem programowania, można wiec obarczyć klieknta duza iloscia pracy zarówno przed jak i po wysłaniu zadania do serwera. Na przykład nie trzeba wysylac zadania, aby odkryc, ze zle podales date lub inny parametr, a sam klient może szybko wykonać cala prace zwiazana ze sporzadeniem wykresu, zamiast czekac, az serwer przygotuje wykres i wysle gotowy obrazek. Otrzymujesz natychmiastowy wzrost predkosci, działania, a ogolny ruch sieci i obciążenie serwerow mogą zostarc zredukowane, przyspieszajac tym samym działanie całego internetu.
Przewaga apletów javy nad programami skryptowymi jest taka ze sa przesylane w postaci skompilowanej, a wiec ich kod zrodlowy jest niedostępny dla klienta. Jeżeli aplet javy jest duzy, to jego pobranie może wymagac dodatkowego czasu.

Aplety są to programiki napisane w Java, które ładuje się odpowiednimi znacznikami na stronę www i które są z poziomu przeglądarki internetowej uruchamiane. Są jednak pewne różnice między aplikacją w Java, a apletem.

Aplety są:
- są inaczej uruchamiane
- nie mają metody main() (są inne pełniące odpowiednie zadania np. init(), paint())
- aplety są podklasami albo JApplet (javax.swing) lub jej klasy bazowej Applet (java.applet).

init() - Inicjalizacja:
Przeglądarka napotykając na metodę init() tworzy wszystkie obiekty jakie ta metoda każe, więc w kod metody daje się takie rzeczy jak kolor tła apletu, przyciski itp.

start() - startująca:
Wykonywana po inicjalizacji lub wywoływana gdy użytkownik wróci na stronę z apletem(może być wykonana wielokrotnie, a inicjalizacja raz ponieważ użytkownik może wracać na stronę z apletem wielokrotnie).

stop() - zatrzymująca:
Wywoływana gdy użytkownik opuści stronę z apletem lub po wywołaniu metody stop().

destroy() - koniec życia apletu (zwolnienie pamięci zajmowanej przez program).

I.14. Zasady usuwania obiektów z pamięci.

Java nie wymaga definiowania destruktorów. Jest tak dlatego, że istnieje mechanizm automatycznego zarządzania pamięcią (ang. garbage collection). Obiekt istnieje w pamięci tak długo, jak długo istnieje do niego jakakolwiek referencja w programie, w tym sensie, że gdy referencja do obiektu nie jest już przechowywana przez żadną zmienną obiekt jest automatycznie usuwany a zajmowana przez niego pamięć zwalniana.
Ponieważ zarządzanie pamięcią jest w Javie zautomatyzowane, nie ma potrzeby definiowania destruktorów. Mamy jednak możliwość deklaracji specjalnej metody finalize, która będzie wykonywana przed usunięciem obiektu z pamięci. Deklaracja takiej metody ma zastosowanie, gdy nasz obiekt np.: ma referencje do urządzeń wejścia-wyjścia i przed usunięciem obiektu należy je zamknąć.

(Z Bąka)
Usuwanie obiektów:
– brak destruktorów – pamięć odśmiecana automatycznie (Garbage Collector)
– gdy klasa rezerwuje pamięć w sposób niestandardowy, należy zwolnić ją ręcznie

```
protected void finalize() throws Throwable {
//...
super.finalize();
}
```
– wymuszenie uruchomienia Garbage Collector
```
System.gc();
System.runFinalization();
```

I.15.Wyjaśnić pojęcia klasy i obiektu.

(z NETA)
Klasa to fundament Javy. Najprościej mówiąc klasa to taka przestrzeń, w której umieszczamy inne elementy, przechowujemy informacje i je przetwarzamy. Klasa definiuje zestaw Metod i Pól dla swoich Obiektów.
(z Bąka)
Przez **obiekt** rozumie się w Javie dynamicznie stworzony egzemplarz jakiejś klasy lub dynamicznie stworzoną tablicę. Żeby to zrównanie egzemplarzy klas i tablic uprawomocnić zadbano, by zarówno egzemplarze klas jak i tablice rozumiały wszystkie metody z klasy Object.

I.16.Omówić, na przykładzie języka Java, najistotniejsze cechy programowania obiektowego.

(Thinking in Java)- nie wiem czy o to chodziło
Według Alana Kay'a:
1. Wszyskto jest obiektem.
2. Program jest zbiorem obiektów, które poprzez wysyłanie komunikatów mówią sobie nawzajem, co robić.
3. Każdy obiekt posiada własną pamięć, na którą składają się inne obiekty.
4. Każdy obiekt posiada swój typ.
5. Wszystkie obiekty danego typu mogą otrzymywać te same komunikaty.

<INTERNET>
Programowanie obiektowe

1. Cechy typowe dla programowania obiektowego
    - abstrakcja - zredukowanie właściwości opisywanego obiektu do najbardziej podstawowych,
    - hermetyzacja danych - dostęp do składowych jest ograniczony za pomocą dobrze

określonego interfejsu,
- dziedziczenie - mechanizm umożliwiający wywodzenie nowych klas z klas już istniejących, wraz z przejmowaniem ich metod,
- polimorfizm - wielopostaciowość, pozwala na wybór metody spośród różnych wersji w zależności od kontekstu.

I.17.Na czym polega wielokrotne wykorzystanie klas?

W Javie istnieje kilka metod na wielokrotne wykorzystanie klas, należą do nich:
- kompozycja: umieszczamy obiekty istniejącej już klasy jako składowe nowej klasy
- agregacja: kompozycja dynamiczna (tworzona w czasie wykonywania programu)
- dziedziczenie: umożliwia rozszerzanie możliwości wcześniej utworzonych klas bez konieczności ich ponownego tworzenia ;Arek

Dziedziczenie pojedyncze i wielokrotne

- Jeżeli nowa klasa jest podklasą tylko jednej klasy bazowej, to taki proces derywacji nazywany jest *dziedziczeniem pojedynczym*.
- Jeżeli nowa klasa ma więcej niż jedną nadklasę, to proces nazywamy *dziedziczeniem wielokrotnym*.

Dziedziczenie jest jedną z podstawowych cech programowania obiektowego. Mechanizm ten umożliwia rozszerzanie możliwości wcześniej utworzonych klas bez konieczności ich ponownego tworzenia
wielokrotne wykorzystanie klas polega na tworzeniu instancji danej klasy.

I.18.Na czym polega „rzutowanie w górę”? Podać przykłady.

(Thinking in Java)
Proces polegający na traktowaniu typu pochodnego tak jakby był swoim typem bazowym nazywamy **rzutowaniem w górę** (ang. upcasting). Rzutowanie używane jest tu w sensie dostosowywania do pewnej formy, w górę zaś odnosi się do sposobu, w jaki zwykle rysuje się diagramy dziedziczenia. Rzutowanie to jest zawsze bezpieczne ponieważ przechodzimy od typu bardziej specyficznego do bardziej ogólnego. Znaczy to, że klasa pochodna jest nadzbiorem klasy bazowej

Poniżej przedstawiony został przykład rzutowania w górę. W poniższym przykładzie obiekty klasy Wind reprezentują pewien typ instrumentów , a zatem klasa Wind jest klasą pochodną Instrument:

```
//Typ wyliczeniowy Note
public enum Note {
MIDDLE_C, C_SHARB, B_FLAT; //itd.
}
//klasa bazowa Instrument
public class Instrument {
public void play(Note n)
{
System.out.println("Instrument.play()");
}
}
//Klasa potomna Wind
public class Wind extends Instrument{
public void play(Note n)
{
System.out.println("Wind.play() " + n );
}
}
//Wywołanie metod z utworzonych klas
public class Music {
public static void tune(Instrument i)
{
i.play(Note.MIDDLE_C);
}
public static void main(String[] args)
{
Wind flute = new Wind();
tune(flute); //Rzutowanie w górę
}
}
```

I.19.Na czym polega „rzutowanie w dół”? Podać przykłady.

Rzutowanie w dół oznacza konwersję typu bazowego na typ pochodny. Rzutowanie z typu ogólnego na typ bardziej szczegółowy. Rzutowanie w dół jest opatrzone błędem. Jeżeli zrzutujemy w dół na niewałaściwy typ, spowodujemy wystąpienie błędu czasu wykonywania, zwanego wyjątkiem. W celu dokonania prawidłowego rzutowania, musimy pamiętać jakiego właściwie jest typu.

rzutowanie w dół (ang. downcasting) – wymaga testowania

```
Object object = (Object) new Samochod(); // rzutowanie w górę
if (object instanceof Samochod) { //sprawdzenie przed rzut.
Samochod samochod = (Samochod) object;//rzutowanie w dół
}
```

I.20.Na czym polega separacja interfejsu od implementacji w programowaniu obiektowym?

Separacja interfejsu od implementacji w programowaniu obiektowym polega na przygotowaniu przez "twórców klas" takich klas, które udostępniają "programiście-klientowi" jedynie to, co dla niego niezbędne, a całą resztę trzymają w ukryciu. To nie jest to samo co polimorfizm, ponieważ tyczy się to jedynie ograniczenia dostępu do zasobów klas, które powinny być chronione przed niewłaściwym wykorzystaniem. Mechanizmy Javy, które umożliwiają stosowanie tej funkcjonalności to słowa kluczowe, służące do ustanowienia rozgraniczeń w klasach: private, protected, public.

l.21 Konstruktor

Konstruktor w programowaniu obiektowym to specjalna metoda danej klasy, wywoływana podczas tworzenia jej instancji. Zadaniem konstruktora jest zainicjowanie obiektu.

- Konstruktor domyślny - można wywołać bez podawania jakichkolwiek parametrów.
- Konstruktor MUSI mieć taką samą nazwę, jak nazwa klasy,
- Konstruktor nie może zwracać żadnego typu (nawet void),
- konstruktor domyślny jest zawsze bezparametrowy,
- konstruktor domyślny można wywołać tylko wtedy, gdy klasa nie posiada żądnego, zaimplementowanego konstruktora, błąd
- Konstruktor może być przeciążany, podczas wywołania konstruktora danej klasy, wywoływane są również wszystkie konstruktory klas nadrzędnych,
- Konstruktor, o ile posiada parametry, może ustawiać pola wartościami, które dostał jako argumenty, jeżeli – konstruktor nie ma żadnych parametrów, wszystkie wartości pól obiektu, będą zainicjalizowane domyślnymi wartościami.
- Konstruktor może być oznaczony dowolnym specyfikatorem dostępu, nawet private (wykorzystywane np. przy implementacji wzorca projektowego Singleton),
- każdy konstruktor ma jako pierwsze wyrażenie albo odwołanie do przeciążonego konstruktora (this()), albo do konstruktora z klasy nadrzędnej (super()), pamiętać należy, że wyrażenia te mogą być dodane przez kompilator w trakcie kompilacji, jeżeli nie zostaną napisane (nie znaczy

to jednak, że zawsze trzeba je jawnie wpisywać w ciało konstruktora)

- Konstruktor nie może mieć zaimplementowane (używać) jednocześnie operatora super() i this(), operator super(), może być wywoływany bez parametrów, oraz z parametrami (dokładnie takimi, jakie ma – konstruktor w klasie nadrzędnej), nie można wywołać metody, lub odwołać się do pola instancji klasy, przed operatorem super(), tylko zmienne lub metody statyczne mogą być użyte jako część wywołania operatora super (np. super(ClassX.PoleStatyczne))
- Interfejsy nie posiadają konstruktorów,
- Konstruktor nie może być wywołany bezpośrednio (tak jak metoda), wywołanie takie może mieć miejsce tylko w innym konstruktorze,
- Konstruktory nie są dziedziczone.
- konstruktory w Javie są niejawnie statyczne

(z Bąka)

Konstruktor jest specjalnym rodzajem metody. Jego nazwa musi być taka sama jak nazwa klasy, w której jest zadeklarowany (po tym kompilator poznaje, że ma do czynienia z konstruktorem, a nie zwykłą metodą). Dla konstruktora nie podaje się typu wyniku. Liczba parametrów konstruktora może być dowolna. Jeśli sami nie zdefiniujemy konstruktora, to kompilator sam wygeneruje bezargumentowy konstruktor domyślny. Dzieje się tak tylko wtedy, jeśli autor klasy nie zdefiniuje żadnego własnego konstruktora.

Słowo kluczowe this. Oznacza ono obiekt, na rzecz którego wykonywana jest metoda. Dzięki użyciu this możemy łatwo wskazać, czy chodzi nam o zmienną obiektową (this.imie) czy o parametr metody (imie).

II. Pytania na ocenę 4.0.

II.1. Na czym polega identyfikacja typu podczas wykonania? Podać przykłady.

Identyfikacja typu potrzebna jest w celu uzyskania dokładnego typu obiektu - czyli **w trakcie rzutowania w dół**:
Ptak p = new Wrobel();
Wrobel w = (Wrobel)p;
Żeby móc poprawnie zrzutować w dół, trzeba wiedzieć jakiego typu był dany obiekt. Uzyskanie informacje w czasie wykonania o typie obiektu to RTTI (run-time type indentification). Żeby sprawdzić czy dany obiekt jest instancją danej klasy, używamy słowa kluczowego **instanceof**:
if (p instanceof Wrobel) {}

II.1. W jaki sposób można porównać dwa obiekty. Podać przykład.

Często chcemy porównać ze sobą dwa obiekty jakiejś klasy. Oczywiście możemy porównać odpowiednie pola klasy i zwrócić wynik, ale.... rozwiązanie idealne polega na zaimplementowaniu w naszej klasie interfejsu Comparable<T>. Dzięki implementacji interfejsu, obiekty powstałe na bazie naszej klasy będą mogły być porównywane ze sobą. Taki sposób porównywania (mówiąc inaczej uporządkowania obiektów) jest wykorzystywany przez Javę np. w kolekcjach. Standardowe klasy np. klasa String również implementują ten interfejs. Dzięki temu porównując dwa łańcuchy tekstowe wiemy który jest "mniejszy", a który "większy" (w tym przypadku wykorzystywane jest uporządkowanie alfabetyczne). Wróćmy do interfejsu Comparable<T>, ma on tylko jedną metodę:

public int compareTo(T obiekt);

Metoda compareTo() może zwracać wartość ujemną, dodatnią albo zero odpowiednio dla obiektów mniejszych, większych lub równych obiektowi przekazanemu jako parametr jej wywołania.

Przyklad:

```
class Student implements Comparable<Student> {
// pola prywatne
private String imie;
private String nazwisko;
private int nrAlbumu;
```

```
// konstruktor klasy
public Student(String imie, String nazwisko, int nrAlbumu) {
this.imie = imie;
this.nazwisko = nazwisko;
this.nrAlbumu = nrAlbumu;
}
// metoda wymagana przez interfejs Comparable<T>
public int compareTo(Student obiekt) {
// zwracamy wynik porównania dwóch pól nazwisko
// (korzystamy z porównania dostępnego w klasie String)
return nazwisko.compareTo(obiekt.nazwisko);
}
// metoda przesłonięta, zwraca nam tekstową reprezentację obiektu
public String toString() {
return (nazwisko + " " + imie + " " + nrAlbumu);
}
}
```

equals sprawdza czy są równe co do wartości
a == czy są równe co do samej referencji

II.2. Zasady definiowania typu wyliczeniowego w języku Java.

(z wykładów)
Po co typ wyliczeniowy?

- Zbiór wartosci okreslony przez uzytkownika
- Symboliczne nazwy wartosci
- Automatyczna kontrola

Definicja typu enum (z neta)
Typ enum jest typem wyliczeniowym, literałem, który jest traktowany jak klasa specjalna zawierająca w swojej definicji wszystkie możliwe do stworzenia instancje, obiekty.

II.3. Co oznacza słowo this . Podać przykłady zastosowania.

Słowo kluczowe **this**. Oznacza ono obiekt, na rzecz którego wykonywana jest metoda. Dzięki użyciu this możemy łatwo wskazać, czy chodzi nam o zmienną obiektową (this.imie) czy o parametr metody

(imie). Inne zastosowanie słowa kluczowego this znajdziemy w przypadku, gdy w jakiejś klasie posiadamy zmienne o takich samych nazwach jak parametry metod, czy konstruktorów.

Przykład:

```
public class Punkt{
 int x;
 int y;
 public Punkt(int x, int y){
 this.x = x;
this.y = y;
}
public void setPunkt(int x, int y){
this.x = x;
this.y = y;
}
}
```

II.4. Co mogą zawierać interfejsy? Jakie są zasady dostępu do elementów interfejsu?

Interfejs w Javie to deklarowany za pomocą słowa kluczowego interface nazwany zbiór deklaracji zawierający:
- publiczne abstrakcyjne metody (bez implementacji),
- publiczne statyczne zmienne finalne (stałe) o ustalonych typach i wartościach.

Implementacja interfejsu w klasie polega na zdefiniowaniu w tej klasie wszystkich metod zadeklarowanych w implementowanym interfejsie.

Ogólna postać definicji interfejsu w języku Java:

```
public interface NazwaInterfejsu
{
typ nazwaZmiennej = wartosc;
 ...
typ nazwaMetody(lista_parametrów);
 ...
}
```

Uwagi:
- modyfikator dostępu public przed słowem interface może nie występować, wówczas interfejs jest

dostępny tylko w bieżącym pakiecie,
- zmienne są zawsze typu static final i mają przypisaną wartość stałą,
- metody są zawsze abstrakcyjne (bez implementacji).
Ogólna postać definicji klasy implementującej interfejs w języku Java:

```
public class NazwaKlasy extends KlasaBazowa implements
NazwaInterfejsu_1, ..., NazwaInterjejsu_n
{
 ...
}
```

Uwagi:
- modyfikator dostępu public przed słowem class może nie występować -
wówczas klasa jest dostępna tylko w bieżącym pakiecie,
- klasa może ale nie musi dziedziczyć inną klasę (słowa extends KlasaBazowa
mogą nie występować)
- klasa może implementować wiele interfejsów,
- klasa musi definiować WSZYSTKIE metody implementowanych interfejsów albo
pozostać klasą abstrakcyjną.
- klasa może zawierać własne (nie będące częścią interfejsu) atrybuty i metody.

II.5. Omówić podstawowe rodzaje kolekcji obiektów w języku Java.

Kolekcja Collection jest grupą odrębnych elementów podlegających określonym regułom. Są to obiekty służące do przechowywania innych obiektów i udostępniające mechanizmy pozwalające na wykonywanie rożnych operacji na zbiorze tych obiektów. Do przechowywania zmiennych prostych oraz obiektów posłużyć mogą tablice, jednak w wielu zadaniach do składowania obiektów i ich przetwarzania tablice okazują się nie wystarczające. W Javie zostały zaprojektowane odpowiednie interfejsy oraz klasy pozwalające na efektywne przechowywanie takich obiektów.

Interfejs Collection jest interfejsem sparametryzowany, w postaci:
interface Collection<Typ>

Klasa Typ określa typ obiektów przechowywanych w kolekcji. Uniemożliwia to dodawanie do kolekcji obiektów innego typu co mogło by powodować błędy. Interfejs ten jest rozszerzeniem Interfejsu Iterable, można do niego zastosować pętlę for-each. Tylko klasy implementujące ten interfejs można przeglądacza pomocą tej pętli. Poniżej przedstawione są najważniejsze metody tego interfejsu:

- add()
- remove()
- size()
- iterator()
- toArray()

Interfejs List jest interfejsem sparametryzowanym postaci:
Interface List<Typ> extends Collection<Typ>

List służy do przechowywania elementów w określonej kolejności. Elementy mogą być pobierane oraz wstawiane na podstawie ich położenia. W przypadku list stosuje się
indeksowanie podobne jak w przypadku tablic.

Klasa Array List implementuje interfejs List i rozszerza klasę AbstractList. ArrayList jest
klasą sparametryzowaną:
Class ArrayList<Typ>

ArrayList implementuje interfejs List jako tablic (rozszerzonej). Pozwala na szybki i swobodny dostęp do elementów. Nie jest dobry rozwiązaniem jeżeli chcemy aby służyła do wstawiania lub usuwania elementów z wnętrza tablicy. Starszą klasą o podobnym zastosowaniu jest klasa Vector.

Klasa LinkedList implementuje interfejs List i rozszerza klasę AbstractList, również jest klasą sparametryzowaną. Dostosowana jest ona do sekwencyjnego przetwarzania. Kolekcję tą mającą strukturę listy można z powodzeniem zastosować do wstawiania i usuwania elementów z początku i końca listy. Dzięki udostępnianym przez siebie metodom klasa ta nadaje się do budowania stosu list jedno i dwukierunkowych oraz kolejek.

Interfejs Set jest interfejsem sparametryzowanym i rozszerzającym interfejs Collection, postaci:
Interface Set <Typ> extends Collection <Typ>

Set oraz SortedSet, definiują zbiory i służą do przechowywania pojedynczych egzemplarzy, ten drugi w postaci uporządkowanej. Nie przyjmują zduplikowanych egzemplarzy, w związku z tym w klasach wywiedzionych z nich musi być zdefiniowana metoda equals().

Klasa HashSet stosowana jest w przypadku zbiorów dla których jest istotny krótki czas lokalizacji. Do przechowywania danych w tej kolekcji służy tablica haszująca. Do wyszukiwania danych stosowane są funkcje mieszające. Funkcje mieszające na podstawie klucza liczą unikatową wartość, tzw. skrót.

Skrót ten służy do indeksowania zawartości.

LinkedHashSet jest klasą podobną do HashSet, przechowuje informacje o kolejności wstawiania danych, odtworzenie tej kolejności wymaga utworzenia iteratora.
TreeSet przechowuje zbiór w postaci uporządkowanej (rosnąco), dokładniej w strukturze drzewa dzięki temu można pobierać elementy uporządkowane. Queue jest interfejsem sparametryzowanym dodanym w Javie 5. Pozwala na wydajne działanie w strukturze kolejek w szczególności kolejek FIFO. Zawiera metody pozwalające na działaniu na elementach początkowych kolejki. PriorityQueue tworzy kolejkę priorytetową, priorytety ustala komputer.

Map (nie jest kolekcją) – interfejs sparametryzowany pozwala na operowanie grupą obiektów typu klucz-wartość. Kontenery Map, tak jak tablice i kolekcje Collection, mogą być łatwo rozbudowywane do wielu wymiarów; wystarczy utworzyć kontener Map, którego elementami są inne kontenery Map. Można zatem w prosty sposób utworzyć kontenery o całkiem pokaźnych strukturach danych. Kontener Map może zwracać zbiór (Set) kluczy, kolekcję (Collection) wartości albo zbiór par.

HashMap implementacja jest oparta na tablicy mieszającej. Zapewnia wstawianie i lokalizację par w stałym czasie.

TreeMap implementacja oparta jest na drzewach. Przechowuje pary w postaci uporządkowanej. Pozwala na dostęp do fragmentu drzewa.
Metody interfejsu Map:
- put()
- get()
- remove()
- size()
- entrySet()

II.6. Omówić cel i zasady stosowania metody finalize().

Istnieje możliwość deklaracji specjalnej metody **finalize**, wykonywanej przed usunięciem obiektu z pamięci. Deklaracja takiej metody jest potrzebna, gdy np.: obiekt ma referencje do urządzeń wejścia-wyjścia i przed usunięciem obiektu należy je zamknąć. Proces zwalniania nieużytków zasobów jest włączany okresowo, uwalniając pamięć zajmowaną przez nieużywane obiekty: przeglądany jest obszar przydzielonej programowi pamięci dynamicznie i zaznaczane obiekty, do których istnieją referencje. Po analizie wszystkich ścieżek referencji do obiektów usuwane są te, do których nie ma żadnej referencji. Mechanizm czyszczenia pamięci działa w wątku o niskim priorytecie synchronicznie

lub asynchronicznie, zależnie od środowiska systemu operacyjnego na którym wykonywany jest program. Program może jawnie uruchomić mechanizm czyszczenia pamięci przez wywołanie metody **System.gc()**. Nie gwarantuje on jednak, że obiekt zostanie usunięty: w systemach, które pozwalają środowisku przetwarzania Javy sprawdzać, kiedy wątek się rozpoczął i przerwał wykonanie innego wątku (np. Windows 95/NT), mechanizm czyszczenia pamięci działa tylko asynchronicznie w czasie bezczynności systemu. Mechanizm czyszczenia pamięci umożliwia obiektowi przed usunięciem "*posprzątanie po sobie*" poprzez wywołanie metody **finalize**. Podczas finalizacji obiekt może zwolnić zasoby systemowe: pliki, gniazdka (ang. *sockets*), referencje do innych obiektów, i inne. Metoda **finalize** jest zdefiniowana w klasie **java.lang.Object**. Klasa musi ją zredefiniować, aby umożliwić finalizację dla zasobów używanych przez obiekty własnego typu.

II.7. Omówić zmienne, metody i klasy ostateczne (final).

Final

- Po klasie oznaczonej jako “final” nie można dziedziczyć.
- Metoda oznaczona słowem “final” nie może zostać nadpisana w sub-klasie (metody abstrakcyjne nie mogą zostać oznaczone jako “final” (ani jako private, czy też static), gdyż tworzone są właśnie po to, aby nadpisać je, w sub-klasach).
- Zmienną oznaczoną słowem “final”, raz zainicjalizowana, nie może zostać już zmieniona. Można zadeklarować zmienną jako final, bez inicjalizowania jej, wtedy pierwsza inicjalizacja będzie możliwa, kolejne już nie.
- Referencja do obiektu oznaczona słowem “final” nie może zostać zmieniona (możliwa jest zmiana wszystkich pól obiektu, ale nie można już zmienić referencji tak, aby pokazywała na inny obiekt, warto zaznaczyć również, iż nie ma finalnych obiektów, jedynie referencje do nich mogą być finalne).
- Parametry metody oznaczone jako finalne (ostateczne), nie mogą zostać zmienione w ciele metody.

II.8. Omówić zasady definiowania klas sparametryzowanych typami.

Możemy:
● podawać je jako typy pól i zmiennych lokalnych,
● podawać je jako typy parametrów i wyników metod,
● dokonywać jawnych konwersji do typów oznaczanych przez nie (ale to będzie tylko ważne na etapie kompilacji, po to by uniknąć błędów niezgodności typów, natomiast nie uzyskamy w fazie wykonania faktycznych konwersji np. zawężających, no bo jak?),
● wywoływać na rzecz zmiennych oznaczanych typami sparametryzowanymi metody klasy

Object (i ew. właściwe dla klas i interfejsów, które stanowią tzw. górne ograniczenia danego parametru typu).

Nie możemy (w definicjach sparametryzowanych klas i metod):
● tworzyć obiektów typów sparametryzowanych (new T() jest niedozwolone, no bo na poziomie definicji generics nie wiadomo co to konkretnie jest T),
● używać operatora instanceOf (z powodu j.w.),
● używać ich w statycznych kontekstach (bo statyczny kontekst jest jeden dla wszystkich różnych instancji typu sparametryzowanego),
● używać ich w literałach klasowych,
● wywoływać metod z konkretnych klas i interfejsów, które nie są zaznaczone jako górne ograniczenia parametru typu (w najprostszym przypadku tą górną granicą jest Object, wtedy możemy używać tylko metod klasy Object).

II.9. Zasady serializacji obiektów.

Serializacja, czy też *marshalling* to technika służąca zapisaniu obiektu danej klasy w postaci ciągu bajtów. Taki ciąg może zostać zachowany na nośniku, przesłany przez sieć, lub pomiędzy procesami, a następnie odtworzony.

Serializacja obiektów w java jest swego rodzaju zrzutem obiektu z pamięci RAM to strumienia bajtowego zapisanego na np dysku twardym. Do tego rodzaju operacji są wykorzystywane klasy strumieniowe : **ObjectOutputStream, ObjectInputStream** oraz odpowiednie metody **void writeObject(Object o)**, **Object readObject()**. Standardowo klasy, które są dostępne w javie od razu posiadają metody do zapisu/odczytu wartości danego typu oraz co **ważne: implementują intefrejs Serializable** np:
writeUTF(String s) / readUTF()

writeFloat(float f) / readFloat()
writeDobule(double d) / read Double()
etc...

II.10. Omówić system wejścia/wyjścia w języku Java.

Klasy bibioteki I/O w Javie są podzielone według wejścia i wyjście. Przez dziedziczenie wszystkie klasy wyprowadzone z klasy InputStream lub Reader mają podstawowe metody read() i write() służące odpowiednio do odczytania i zapisania jednego lub tablicy bajtów. Metody te jednak rzadko używane

są bezpośrednio - istnieją po to, aby mogły skorzystać z nich inne klasy dostarczające bardziej użytecznych interfejsów.