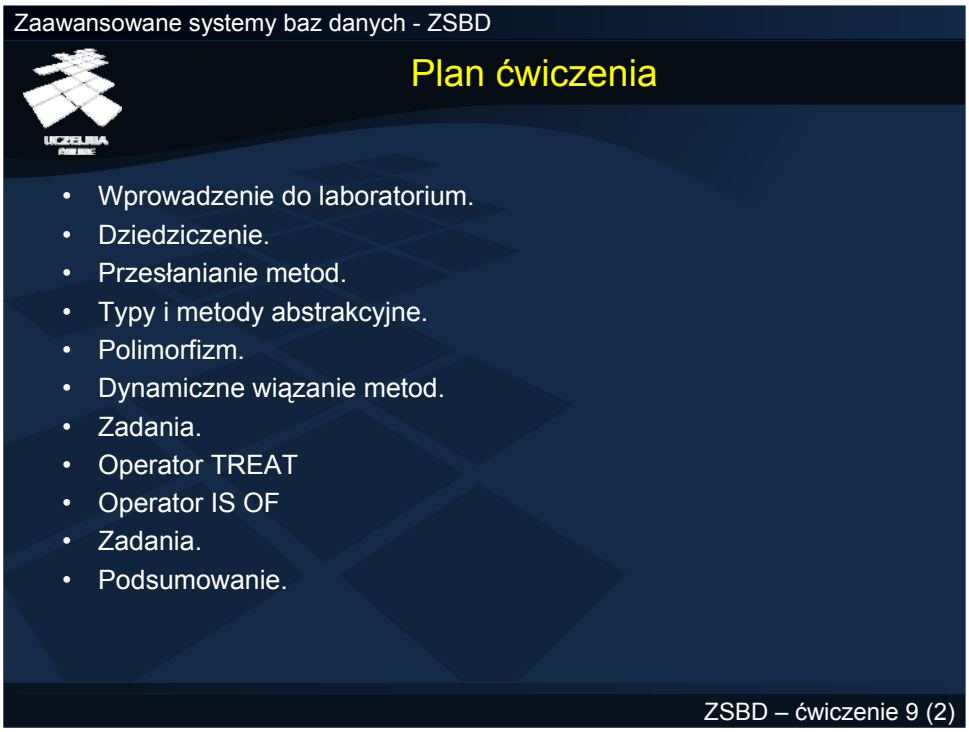
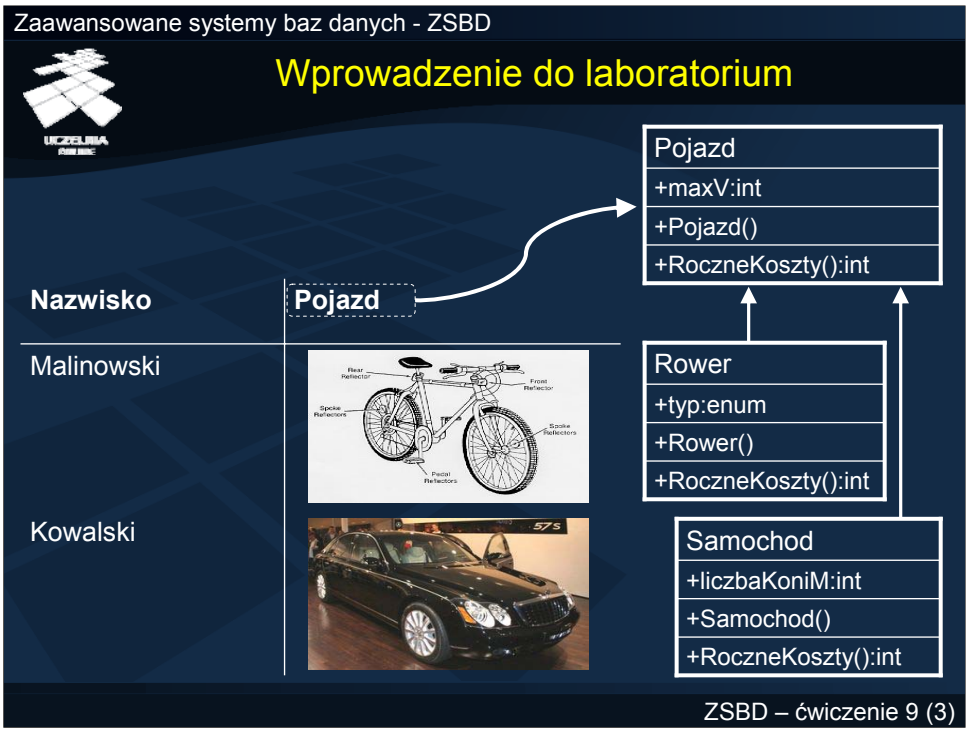
Na dotychczasowych ćwiczeniach poznaliście Państwo składnię poleceń pozwalających na: tworzenie złożonych typów (obiektów, kolekcji), składowanie wartości tych typów w tabelach i wykonywanie zapytań odwołujących się do takich wartości. Typy obiektowe potraktowano jednak w uproszczony sposób. Były to w zasadzie rekordy, z tą jedną różnicą, że można było zaimplementować w nich metody. Nie omówiono zatem pewnych charakterystycznych cech typów obiektowych, które stanowią o sukcesie modelu obiektowego. Celem ćwiczenia 9 jest przedstawienie państwu tych cech. Dowiecie się Państwo jak można utworzyć hierarchię typów obiektowych, jak można przesłonić metodę oraz zadeklarować metodę abstrakcyjną, oraz jak można wykonywać zapytania do tabel przechowujących obiekty różnych podtypów.

Wymagania:

Do wykonania ćwiczenia konieczna jest dobra znajomość języka SQL i przeciętna PL/SQL oraz tematyka omawiana na siódmych i ósmych ćwiczeniach z ORSZBD.

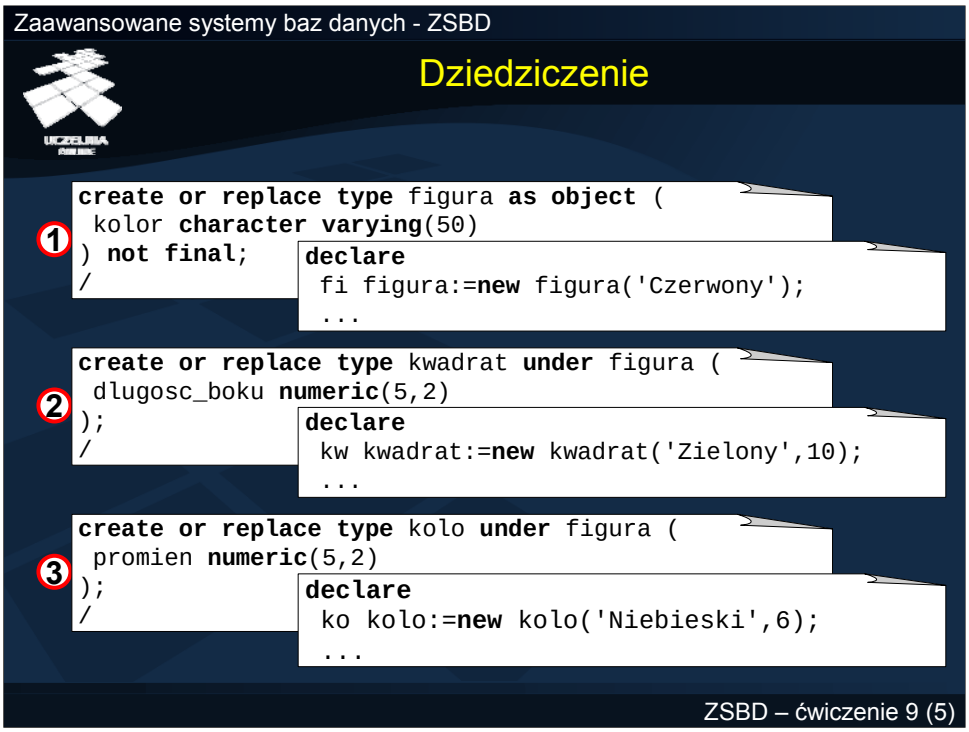


Ćwiczenie rozpocznie się od wprowadzenia motywacji będącej przyczyną implementacji pewnych cech modelu obiektowego w modelu obiektowo – relacyjnym. Następnie, omówiona zostanie składnia poleceń pozwalająca na: tworzenie typów dziedziczących ze zdefiniowanych wcześniej typów obiektowych, przesłanianie istniejących metod, deklarowanie typów i metod abstrakcyjnych, oraz omówiony zostanie polimorfizm i związane z nim dynamiczne wiązanie metod, oraz sposób wykorzystania tych cech przy konstrukcji zapytań. Po omówieniu wyżej wymienionych tematów przedstawione zostaną zadania do samodzielnego wykonania. Następnie, przedstawimy Państwu operatory TREAT i IS OF pozwalające na wykonywanie zapytań do tabel przechowujących obiekty różnych podtypów. Działanie tych operatorów przećwiczycie państwo na kolejnych zadaniach do samodzielnego wykonania. Ćwiczenie zakończymy slajdem podsumowującym przedstawiony materiał.



Jedną z ważniejszych cech modelu obiektowego jest możliwość tworzenia typów obiektowych stanowiących specjalizację innych typów. Typy stanowiące specjalizację (inaczej podtypy) dziedziczą wszystkie metody i atrybuty swoich nadtypów, ale mogą również wprowadzać nowe atrybuty i metody, oraz wprowadzać modyfikacje do implementacji odziedziczonych metod. W modelu obiektowo-relacyjnym również istnieje możliwość tworzenia specjalizacji typów obiektowych. Daje to niesłychaną elastyczność budowanych schematów tabel oraz niezwykle łatwą budowę i późniejsze rozszerzanie funkcjonalności aplikacji. Rozważmy przykładową tabelę pokazaną na slajdzie opisującą posiadane przez pracowników firmy pojazdy, którymi docierają do miejsca pracy. Posiada ona dwa atrybuty: NAZWISKO i POJAZD. Pierwszy atrybut określa nazwisko pracownika, a drugi jest typu obiektowego POJAZD i charakteryzuje pojazd pracownika. W typie obiektowym POJAZD zadeklarowano jedynie pola opisujące wszystkie pojazdy niezależnie od ich faktycznego typu (tutaj jest to maksymalna prędkość). Każdy typ pojazdu posiada jednak charakterystyczne dla siebie wartości. W sytuacji, gdyby wykorzystany był model relacyjny, to należałoby albo utworzyć dla każdego typu pojazdu osobną tabelę, albo w jednej tabeli przechowywać wszystkie dane dotyczące wszystkich typów pojazdów. Obydwa rozwiązania są kłopotliwe i prowadzą do konieczności rozwiązania różnych problemów przy konstrukcji zapytań. Dodatkowe problemy powstają w sytuacji gdy zachodzi konieczność uwzględnienia kolejnego typu pojazdu, który nie został przewidziany w momencie tworzenia aplikacji. W model obiektowo – relacyjnym, możliwe jest utworzenie podtypów typu pojazd, które przechowują dane charakterystyczne dla konkretnego typu pojazdu. Jak łatwo zauważyć, aby dodać nowy typ pojazdu, nie trzeba wprowadzać żadnych modyfikacji do istniejącego schematu tabel. Wystarczy jedynie zdefiniować nowy typ obiektowy dziedziczący z typu POJAZD. Załóżmy, że aplikacja korzystająca z tabeli przedstawionej na slajdzie oblicza dla każdego pracownika jego roczne koszty utrzymania pojazdu.

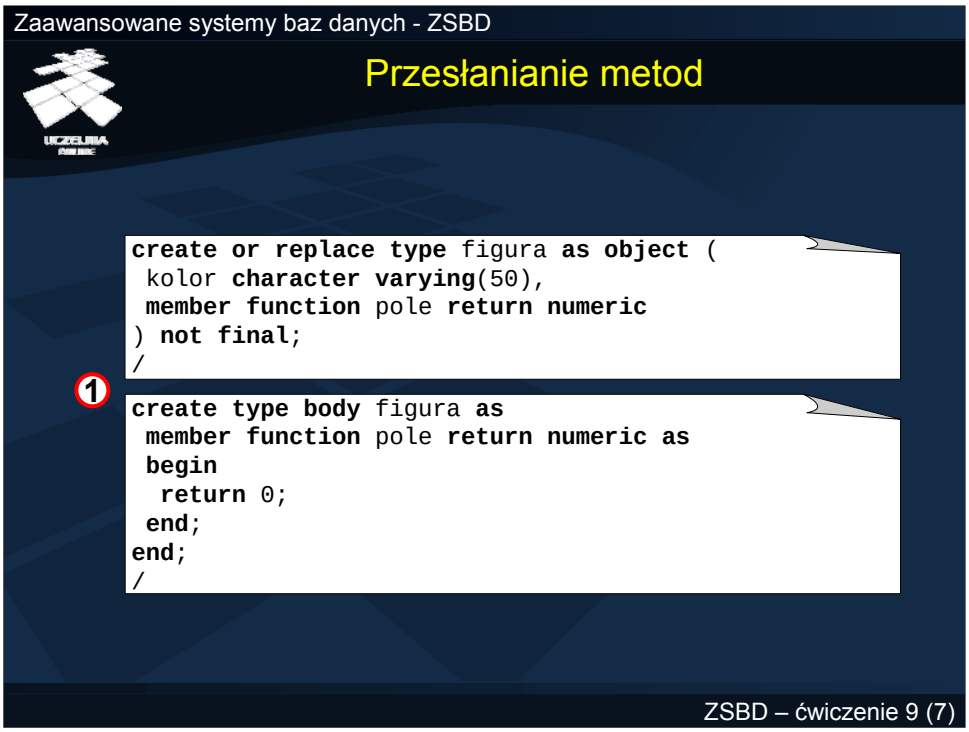
W modelu obiektowo relacyjnym można zaimplementować w typie POJAZD, i jego podtypach, metodę RoczneKoszty, której implementacja jest charakterystyczna dla danego typu pojazdu i oblicza potrzebną aplikacji wartość. Dzięki temu, w sytuacji gdy dodawany będzie do aplikacji kolejny typ pojazdu, nie będzie trzeba modyfikować zapytań obliczających roczny koszt utrzymania, gdyż będą one jedynie aktywować metody charakterystyczne dla danego podtypu.



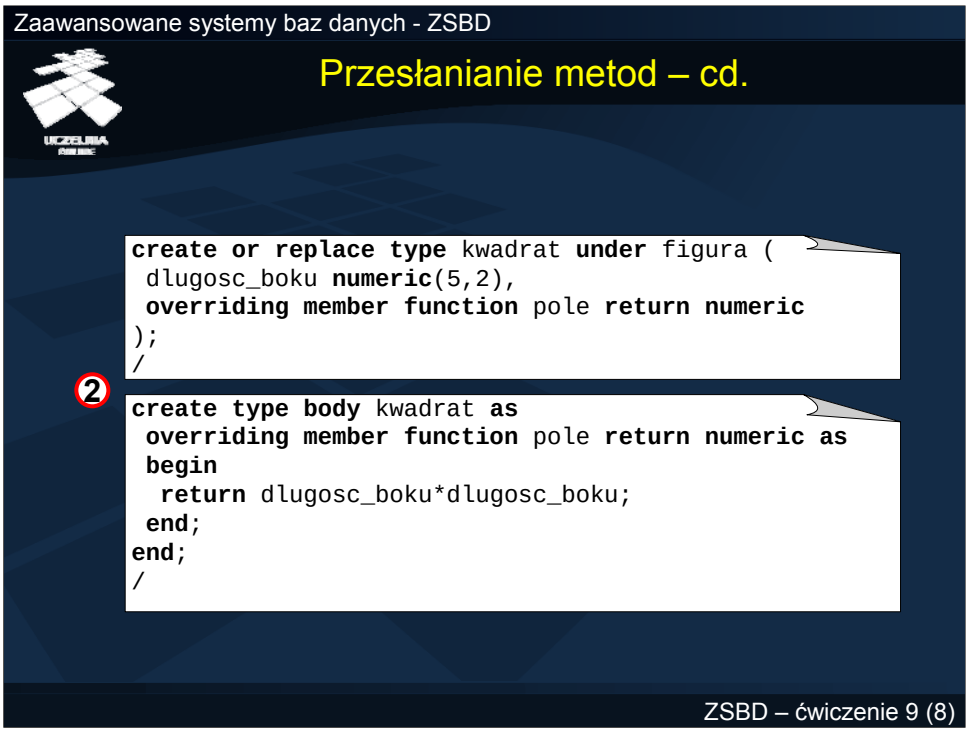
Na slajdzie pokazano sposób utworzenia przykładowej hierarchii typów. Typy stanowiące korzeń hierarchii dziedziczenia tworzy się tak, jak przedstawiono już wielokrotnie na poprzednich zajęciach. Przykład (1) pokazuje polecenie tworzące typ FIGURA o atrybucie KOLOR, który będzie stanowić korzeń naszej przykładowej hierarchii. Warto tutaj zwrócić uwagę na jeden nowy element, który nie pojawiał się wcześniej. Na końcu polecenia tworzącego typ FIGURA dodano słowa kluczowe NOT FINAL. Aby wytłumaczyć znaczenie tych słów należy wprowadzić jeszcze jedną cechę modelu obiektowo-relacyjnego zaimplementowanego w SZBD Oracle. Otóż w modelu tym możliwe jest zablokowanie możliwości tworzenia specjalizacji wybranych typów. Domyślnie typ obiektowy jest tworzony jest z zablokowaną możliwością tworzenia podtypów, chociaż można wskazać to jawnie umieszczając słowo kluczowe FINAL na końcu polecenia tworzącego typ. Podobną funkcjonalność ma w języku JAVA słowo kluczowe final. Aby istniała możliwość tworzenia podtypów typu obiektowego w SZBD Oracle, należy to jawnie nakazać za pomocą słów kluczowych NOT FINAL umieszczonych na końcu polecenia tworzącego typ. W sytuacji, gdy typ, dla którego chcemy utworzyć podtyp został utworzony wcześniej ze słowem FINAL, bądź bez żadnego słowa kluczowego określającego możliwość dziedziczenia, można zmienić jego definicję za pomocą polecenia ALTER TYPE nazwa_typu NOT FINAL;

Polecenia tworzące podtypy (2) i (3) różnią się nieznacznie od polecenia tworzącego korzeń hierarchii typów. Różnicą jest tutaj to, iż zastąpiono słowa kluczowe AS OBJECT słowem kluczowym UNDER po którym podano nazwę nadtypu tworzonego typu. Nowe podtypy reprezentują konkretne kształty figur, czyli kwadrat i koło. Każdy z nowych podtypów posiada atrybuty charakterystyczne dla reprezentowanego przez nie kształtu: DLUGOSC_BOKU dla typu KWADRAT i PROMIEN dla typu KOLO.

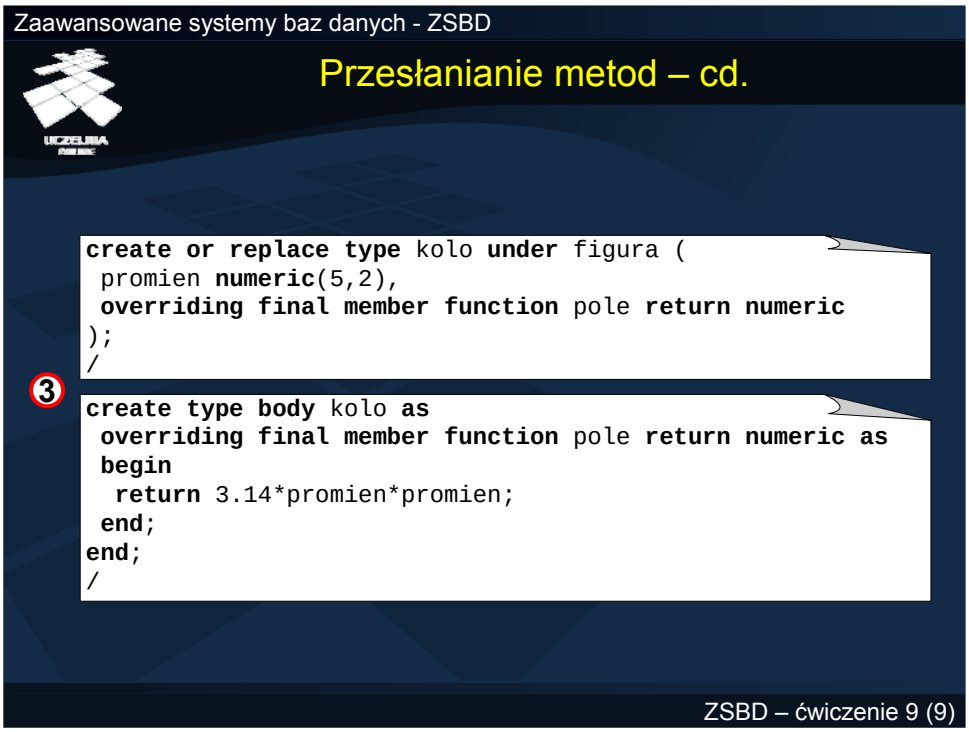
Ponieważ podtyp dziedziczy wszystkie atrybuty nadtypu, to powstaje problem kolejności, w jakiej należy podawać wartości atrybutów w konstruktorze atrybutowym typu obiektowego. Obok każdego polecenia tworzącego typ obiektowy przedstawiono również przykładowe wywołanie konstruktora atrybutowego. Jak można łatwo zauważyć, najpierw podawane są wartości atrybutów z nadtypu, a dopiero później nowe atrybuty wprowadzone w podtypie. W sytuacji gdy byłoby więcej typów na ścieżce od korzenia, do typu, którego konstruktor jest wywoływany, to najpierw należałoby podawać wartości atrybutów korzenia, a następnie wartości atrybutów kolejnych typów na ścieżce w dół hierarchii dziedziczenia.



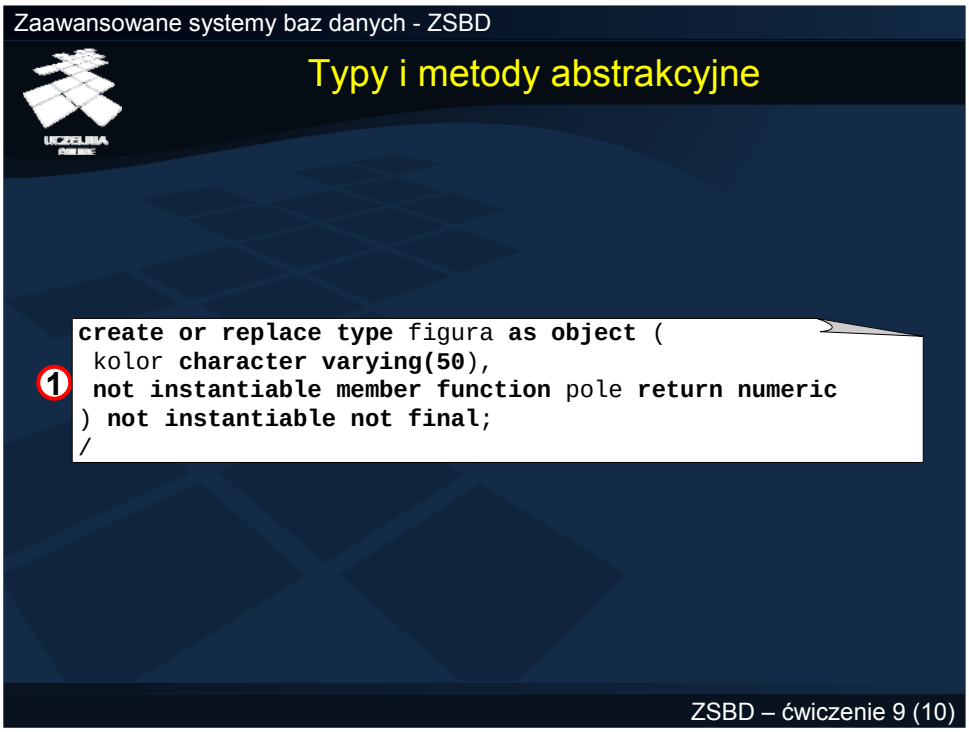
Jak wspominano już przy okazji omawiania poprzedniego slajdu, w modelu obiektowym oraz w modelu obiektowo – relacyjnym istnieje możliwość modyfikacji implementacji metod odziedziczonych z nadtypu. Taką modyfikację nazywa się przesłanianiem. Ten i kolejne dwa slajdy przedstawiają zmodyfikowaną przykładową hierarchię typów z poprzedniego slajdu, którą wzbogacono o deklaracje i implementacje metod. W typie FIGURA (1) zadeklarowano metodę POLE, której zadaniem jest obliczenie pola powierzchni figury. Ponieważ nie znamy właściwego kształtu figury reprezentowanej przez obiekt typu FIGURA, metodę POLE zaimplementowano tak, aby zawsze zwracała wartość zero. ...



... Typy dziedziczące z typu FIGURA: KWADRAT i KOLO (patrz kolejny slajd) również deklarują metodę POLE, dla której posiadają własną implementację zwracającą konkretną wartość. Zacznijmy od analizy typu KWADRAT. Deklaracja metody POLE różni się tutaj od deklaracji tej metody w typie FIGURA tym, że deklarację rozpoczęto od słowa kluczowego OVERRIDING, które sygnalizuje, że deklarowana metoda będzie przesłaniać metodę z nadtypu. Użycie słowa kluczowego OVERRIDING jest obowiązkowe, a jego pominięcie jest zgłaszane jako błąd. Jeżeli teraz przeanalizujemy ciało typu KWADRAT, możemy zauważyć, że również przy implementacji metody POLE należy użyć słowa OVERRIDING. ...

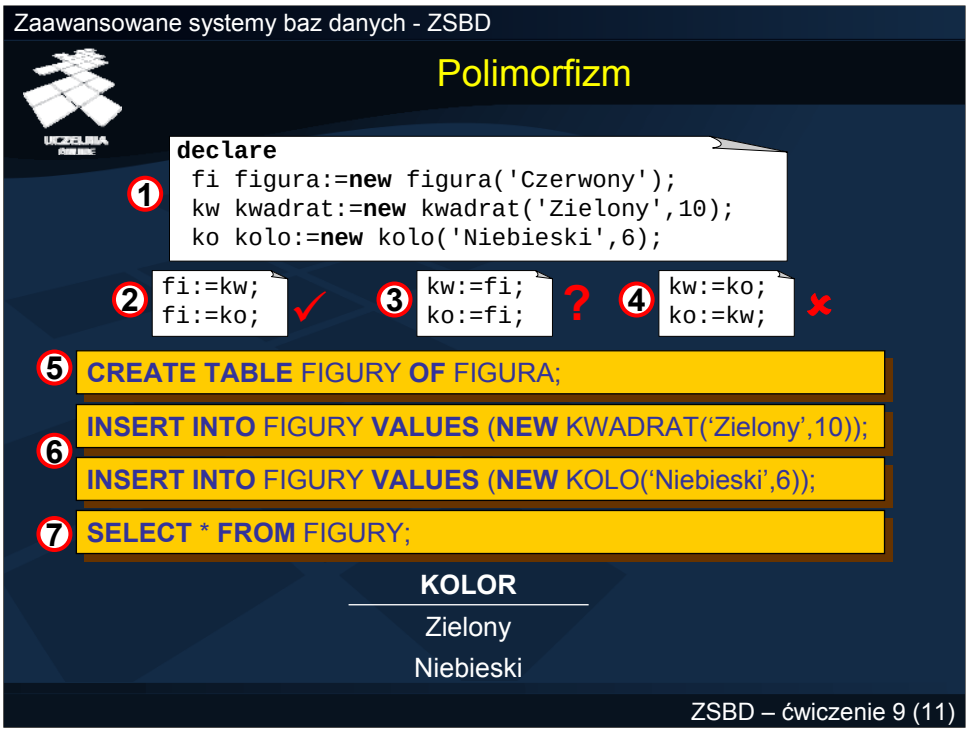


... Deklaracja metody POLE w typie KOLO (3) zawiera jeszcze jedno słowo kluczowe: FINAL. Znaczenie słowa FINAL użytego w tym miejscu jest podobne do tego, kiedy zostanie ono użyte za definicją typu obiektowego. Ogranicza ono możliwość przesłaniania tej implementacji metody w podtypach typu KOLO. Jeżeli spojrzymy na ciało typu możemy zauważyć, że słowa FINAL użyto również przy implementacji metody POLE.



W przykładzie przedstawionym na poprzednich slajdach, metoda POLE w typie FIGURA posiada bezużyteczną implementację. Ponieważ nie było wiadomo jak policzyć pole figury, której kształt jest nie znany, zaimplementowano tą metodę tak, aby zawsze zwracała zero. Z drugiej strony i tak wiadomo, że z metody tej nie będziemy korzystać, bo metoda, która nic nie liczy nie jest w żaden sposób przydatna. Można zatem w ogóle zrezygnować z implementacji tej metody i pozostawić jedynie jej deklarację w typie FIGURA. Metody, których deklaracja stanowi jedynie zapowiedź, że w którymś z podtypów pojawi się ich implementacja, nazywa się metodami abstrakcyjnymi. Typy obiektowe, w których zadeklarowano metody abstrakcyjne, nazywa się typami abstrakcyjnymi. Typy abstrakcyjne z oczywistych względów nie mogą posiadać instancji, a zatem nie jest możliwe tworzenie obiektów typów abstrakcyjnych. Polecenie (1) tworzy abstrakcyjny typ FIGURA, z abstrakcyjną metodą POLE. Zacznijmy od analizy deklaracji metody. Słowo kluczowe MEMBER rozpoczynające poprzednio deklarację metody poprzedzono słowami kluczowymi NOT INSTANTIABLE, które oznaczają, że metoda ta nie będzie implementowana. Podobną funkcjonalność ma np. słowo kluczowe abstract w języku JAVA. Ponieważ typ FIGURA zawiera abstrakcyjną metodę, to musi być zdefiniowany jako typ abstrakcyjny. Jest to wykonywane poprzez umieszczenie słów kluczowych NOT INSTANTIABLE na końcu polecenia tworzącego typ. Przesłanianie metod abstrakcyjnych odbywa się w taki sam sposób jak zwykłych metod posiadających implementację. W związku z tym, polecenia (2) i (3) z poprzednich dwóch slajdów tworzące typy KWADRAT i KOLO, oraz polecenia tworzące ciała tych typów są poprawne również, jeżeli typ FIGURA jest utworzony za pomocą polecenia pokazanego na tym slajdzie.

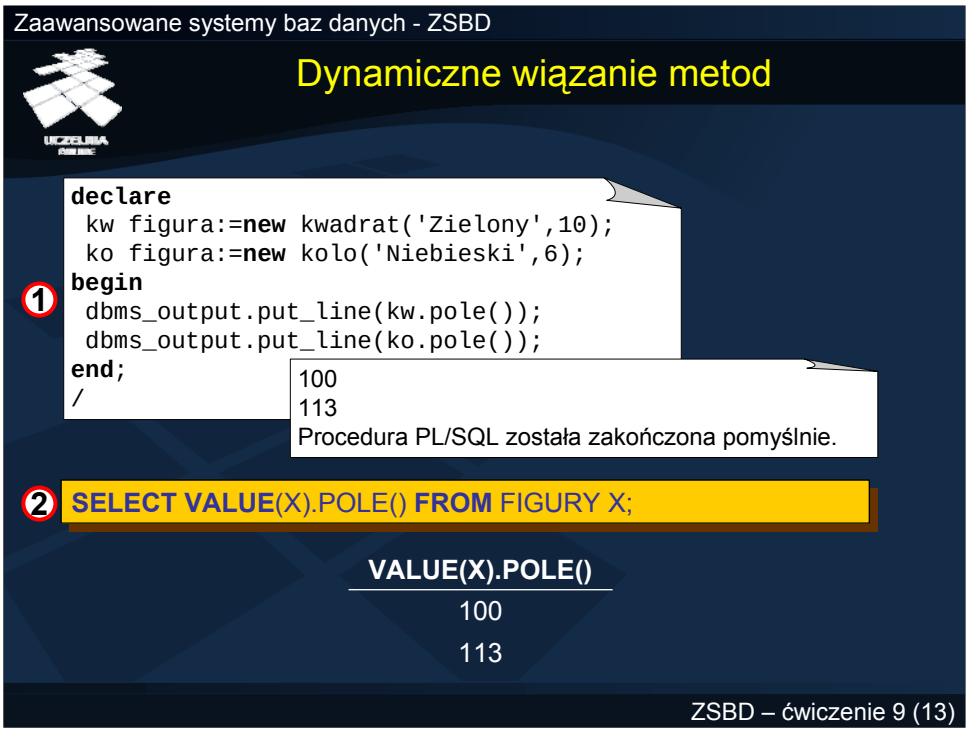
Warto tutaj wspomnieć jeszcze o jednej rzeczy. Otóż teoretycznie możliwe jest zadeklarowanie typu, który byłby abstrakcyjny (NOT INSTANTIABLE) i równocześnie nie byłoby możliwe tworzenie jego podtypów (FINAL). Typy takie nie posiadają jednak żadnego praktycznego znaczenia i próba utworzenia takiego typu kończy się błędem.



Jedną z własności modelu obiektowego i obiektowo-relacyjnego jest polimorfizm. Polega on na tym, że do zmiennej typu obiektowego można przypisać obiekt jej typu, bądź dowolnego podtypu. Przykładowo, do zmiennej typu FIGURA można przypisać obiekty typu FIGURA (o ile nie jest to typ abstrakcyjny), oraz obiekty dowolnego podtypu typu FIGURA, na przykład obiekty typu KWADRAT albo KOLO. Poprzez takie zmienne jednak mamy dostęp jedynie do atrybutów zadeklarowanych w typie FIGURA. Dostęp do atrybutów specyficznych dla podtypu nie jest możliwy w sposób bezpośredni. Przeanalizujmy przykład na slajdzie. Fragment programu na przykładzie (1) pokazuje deklarację zmiennych typów FIGURA, KWADRAT i KOLO, oraz utworzenie obiektów typów KWADRAT i KOLO. Przykład (2) pokazuje dopuszczalne przypisania: do zmiennej typu FIGURA można przypisać obiekty typu KWADRAT albo KOLO. W drugą stronę (3) takie przypisanie nie zawsze jest możliwe. Przykładowo, jeżeli próbujemy do zmiennej typu KWADRAT przypisać zawartość zmiennej typu FIGURA, to może się zdarzyć, że w zmiennej tej będzie się krył obiekt typu KWADRAT i wówczas takie przypisanie byłoby poprawne. Jednak może się tam kryć również obiekt typu KOLO i wówczas przypisanie nie jest możliwe. Aby rozwiązać ten problem stosuje się odpowiednią konstrukcję, która zostanie wprowadzona później. Przypisania dokonywane w sposób przedstawiony na (3), czyli bez użycia tej konstrukcji, nie są dopuszczalne i kończą się błędem. Przypisania pomiędzy zmiennymi typów nie znajdujących się na jednej ścieżce w hierarchii dziedziczenia nie są nigdy dozwolone (4).

Przykłady (1)(2)(3) i (4) pokazywały polimorficzne przypisania w języku PL/SQL, ale powyższa dyskusja odnosi się również do tabel składujących obiekty i języka SQL. Polecenie (5) tworzy tabelę obiektową typu FIGURA. Taką tabelę można traktować jako zbiór zmiennych typu FIGURA. Możliwe są zatem polimorficzne przypisania do tych „zmiennych” obiektów podtypów typu FIGURA.

Demonstrują to polecenia INSERT (6), które wstawiają to tabeli FIGURY obiekty typów KWADRAT i KOLO. Jak wspominano wcześniej, poprzez zmienne nadtypu dostępne są jedynie atrybuty, które są zadeklarowane w nadtypie, natomiast wszelkie atrybuty charakterystyczne dla podtypów nie są dostępne w bezpośredni sposób. Odnosi się to również do zapytań do tabel składujących obiekty. Zapytanie (7) odczytuje wartość wszystkich dostępnych atrybutów z tabeli FIGURY. Jak łatwo zauważyć na wyniku zapytania, odczytany został jedynie KOLOR, który został zadeklarowany w typie FIGURA. Atrybuty DLUGOSC_BOKU i PROMIEN charakterystyczne dla typów odpowiednio KWADRAT i KOLO nie zostały odczytane.



Tym co stanowi o sile modelu obiektowego jest dynamiczne wiązanie metod. W praktyce znaczy to, że jeżeli aktywujemy metodę poprzez zmienną nadtypu, to zostanie aktywowana metoda zaimplementowana w ciele typu obiektu, który faktycznie został do tej zmiennej przypisany. Przykładowo, jeżeli do zmiennej typu FIGURA zostanie przypisany obiekt typu KWADRAT, a następnie poprzez tą zmienną zostanie aktywowana metoda POLE, to faktycznie zostanie aktywowana metoda POLE zaimplementowana w typie KWADRAT. Powyższą dyskusję ilustrują przykłady pokazane na slajdzie. Przykład(1) pokazuje dynamiczne wiązanie metod w jeżyku PL/SQL. Do zmiennych KW i KO typu FIGURA przypisywane są obiekty typu KWADRAT i KOLO. Następnie, poprzez te zmienne, aktywowane są metody POLE, a ich wyniki są wyświetlane na konsoli. Jak łatwo zauważyć, aktywowanie metody POLE poprzez zmienną przechowującą kwadrat o boku 10, powoduje wykonanie kodu obliczającego pole kwadratu (10*10=100). Z kolei aktywowanie metody POLE poprzez zmienną przechowującą koło o promieniu 6, powoduje wykonanie kodu obliczającego pole koła (3.14*6*6=113.04). Z analogicznym zjawiskiem mamy do czynienia podczas realizacji zapytania (2). W zapytaniu tym odczytywane są wszystkie obiekty z tabeli obiektowej FIGURY i dla każdego z tych obiektów aktywowana jest metoda POLE, a jej wynik zwracany jest w tabeli wynikowej zapytania. Jak łatwo zaważyć, podobnie jak poprzednio, wykonane zostały implementacje metody POLE charakterystyczne dla typu obiektu.

Zadanie (1)

- Zbuduj hierarchię typów obiektowych reprezentujących zwierzęta: typ DRAPIEZNIK i dziedziczące z niego LEW i REKIN. W typie obiektowym DRAPIEZNIK zdefiniuj atrybut LICZBA_OFIAR i abstrakcyjną metodę ILE_UPOLOWANO. Metodę zaimplementuj w odpowiednich podtypach. Metoda ta powinna być funkcją, która zwraca komunikat zależny od aktualnego typu obiektowego i liczby ofiar zapisanej w obiekcie. Działanie metod przetestuj za pomocą programu przedstawionego na kolejnym slajdzie.

Zadanie (1) – cd.

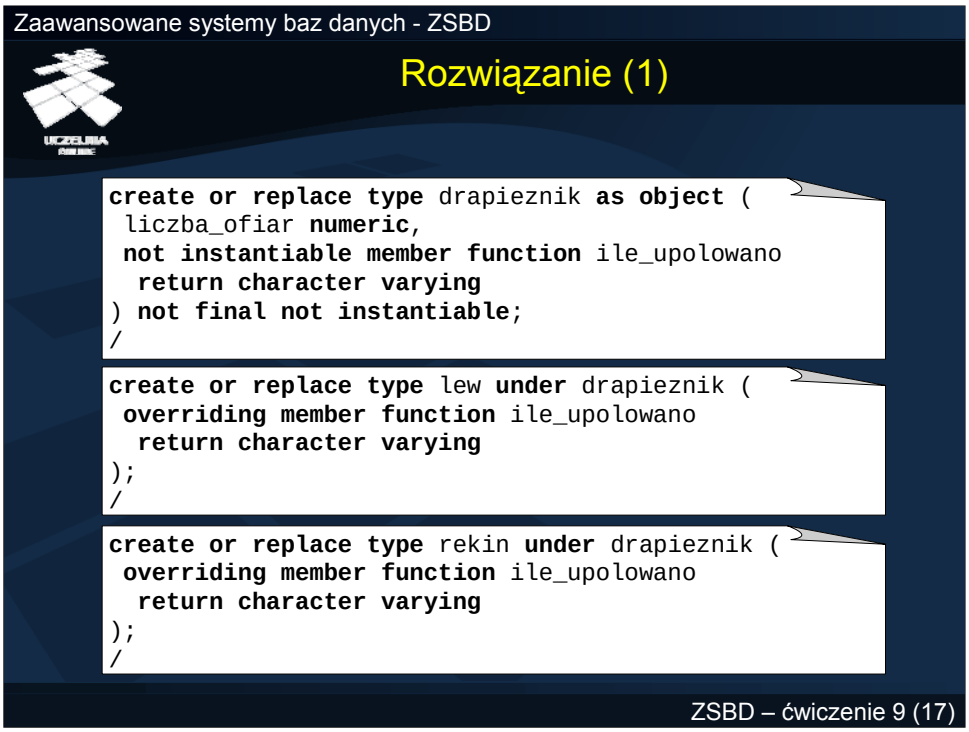
```
declare
 l drapieznik:=lew(5);
 r drapieznik:=rekin(3);
begin
 dbms_output.put_line(l.ile_upolowano());
 dbms_output.put_line(r.ile_upolowano());
end;
/
```

```
Krol lew upolowal 5 zwierzat.
Rekin zjadl już 3 ryb.
Procedura PL/SQL została zakończona pomyślnie.
```

Zadanie (2)

- Utwórz tabelę obiektową DRAPIEZNIKI, która przechowuje obiekty typu DRAPIEZNIK. Zapisz do tej tabeli kilka obiektów typu LEW oraz REKIN z różnymi liczbami ofiar. Wykonaj poniższe zapytanie do tej tabeli. Co zauważyłeś?

```
SELECT VALUE(D).ILE_UPOLOWANO()
FROM DRAPIEZNIKI D;
```



Slajd pokazuje rozwiązanie zadania (1), którego treść przytoczono poniżej. Rozwiązanie jest kontynuowane na kolejnym slajdzie.

Zbuduj hierarchię typów obiektowych reprezentujących zwierzęta: DRAPIEZNIK i dziedziczące z niego LEW i REKIN. W typie obiektowym DRAPIEZNIK zdefiniuj atrybut LICZBA_OFIAR i abstrakcyjną metodę ILE_UPOLOWANO. Metodę zaimplementuj w odpowiednich podtypach. Metoda ta powinna być funkcją, która zwraca komunikat zależny od aktualnego typu obiektowego i liczby ofiar zapisanej w obiekcie. Działanie metod przetestuj za pomocą programu przedstawionego na kolejnym slajdzie.

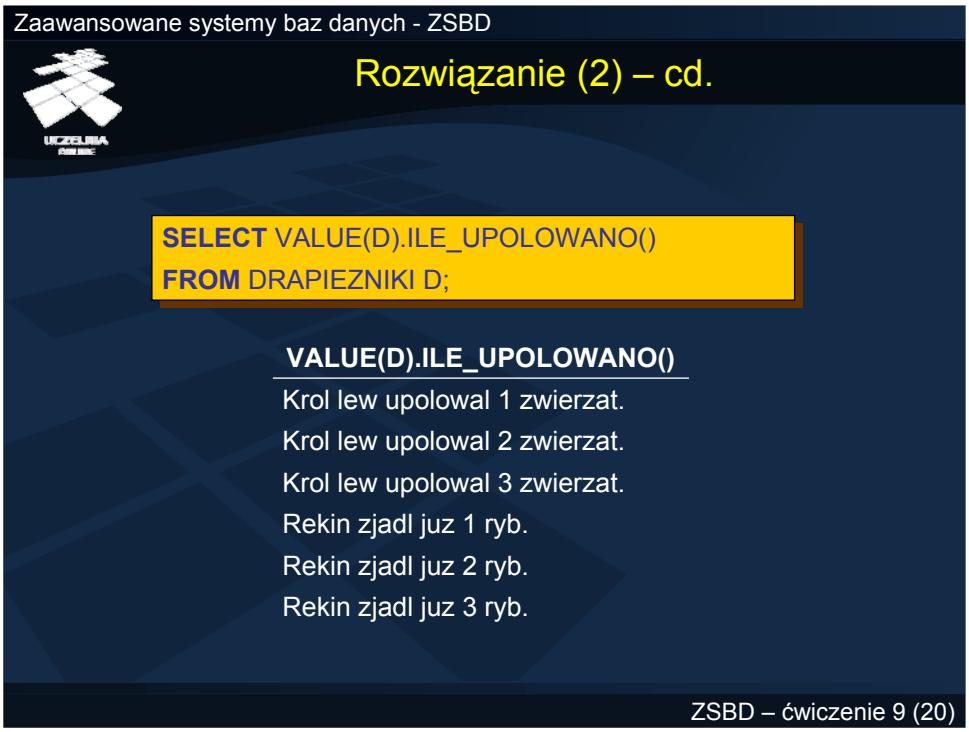
Rozwiązanie (1) – cd.

```
create or replace type body lew as
 overriding member function ile_upolowano
  return character varying as
 begin
  return 'Krol lew upolowal '||liczba_ofiar||' zwierzat.';
 end;
end;
/
```

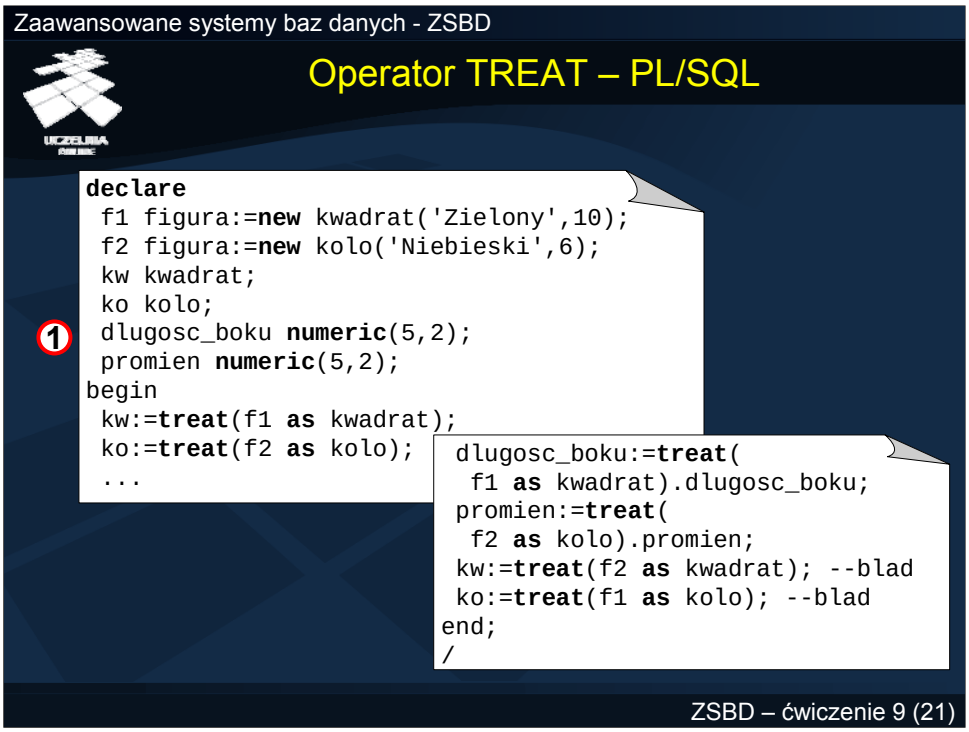
```
create or replace type body rekin as
 overriding member function ile_upolowano
  return character varying as
 begin
  return 'Rekin zjadl juz '||liczba_ofiar||' ryb.';
 end;
end;
/
```

Slajd pokazuje rozwiązanie zadania (2), którego treść przytoczono poniżej.
Rozwiązanie jest kontynuowane na kolejnym slajdzie.

Utwórz tabelę obiektową DRAPIEZNIKI, która przechowuje obiekty typu DRAPIEZNIK.
Zapisz do tej tabeli kilka obiektów typu LEW oraz REKIN z różnymi liczbami ofiar...

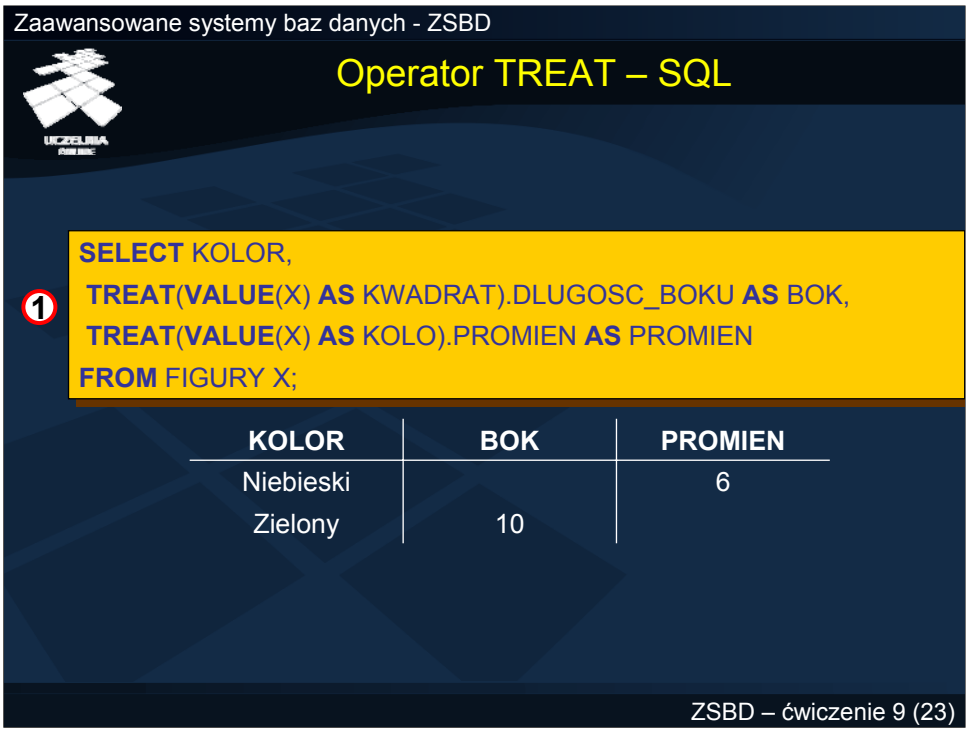


...Wykonaj poniższe zapytanie do tej tabeli. Co zauważyłeś?

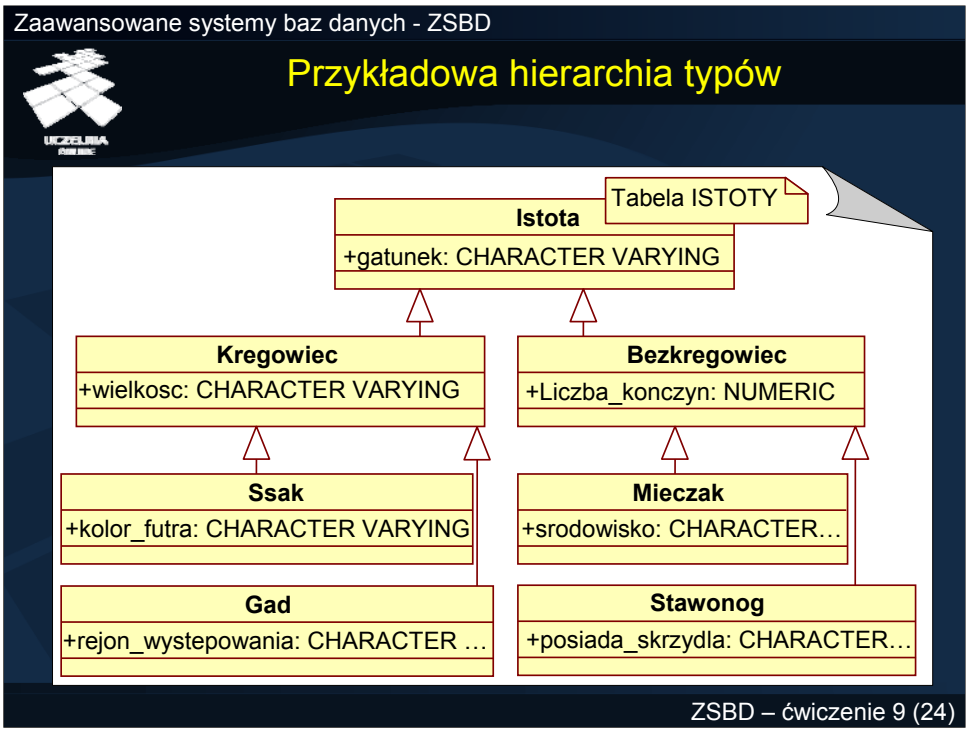


Jak wspominano wcześniej, poprzez zmienne nadtypu dostępne są jedynie atrybuty, które zadeklarowano w nadtypie, natomiast wszelkie atrybuty charakterystyczne dla podtypów nie są dostępne w bezpośredni sposób. Obecnie przedstawiony zostanie sposób uzyskania dostępu do pól charakterystycznych dla podtypów. Do uzyskania takiego dostępu należy użyć operatora rzutowania w dół hierarchii dziedziczenia o nazwie TREAT. Składnia operatora jest następująca: TREAT (zmienna AS typ_obiektowy). Operator zmienia typ zmiennej na typ podany po słowie kluczowym AS. Jeżeli taka zamiana nie jest możliwa, to są dwa możliwe zachowania. W języku PL/SQL zgłaszany jest wyjątek, a w języku SQL operator zwraca wartość NULL. Program na przykładzie (1) demonstruje użycie operatora TREAT w języku PL/SQL. Program rozpoczyna się od deklaracji zmiennych F1 i F2 typu FIGURA, którym przypisywane są odpowiednio obiekty typu KWADRAT i KOLO. Dodatkowo, deklarowane są zmienne KW i KO, odpowiednio typów KWADRAT i KOLO oraz zmienne typu liczbowego o nazwach DLUGOSC_BOKU i PROMIEN. Pierwsze instrukcje programu demonstrują przypisanie do zmiennej podtypu obiektu zapisanego w zmiennej nadtypu. Są to przypisania, które zostały oznaczone znakiem zapytania na slajdzie pod tytułem „Polimorfizm”. W zmiennej F1 typu FIGURE zapisano obiekt typu KWADRAT. Za pomocą operatora TREAT typ zmiennej F1 jest zmieniany na typ KWADRAT i w rezultacie przypisanie wartości wyrażenia do zmiennej KW typu KWADRAT jest możliwe. Działanie kolejnego wiersza programu jest analogiczne. Kolejne dwa wiersze demonstrują dostęp do pól specyficznych dla podtypu za pomocą operatora TREAT. Ponieważ wartością operatora TREAT jest obiekt, to możliwe jest użycie operatora kropkowego w celu odczytania wartości atrybutu. W ten sposób odczytywane są wartości atrybutów DLUGOSC_BOKU i PROMIEN, które są następnie zapisywane do odpowiednich zmiennych.

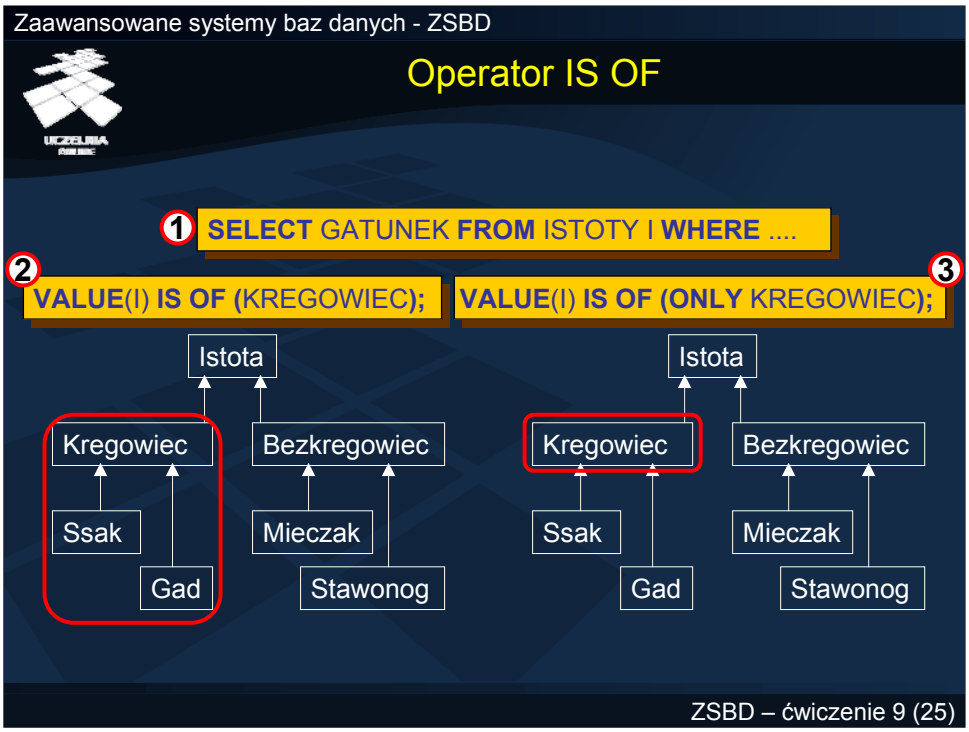
Ostatnie dwie linijki programu powodują błąd wykonania. W obu przypadkach mamy tutaj do czynienia z niedopuszczalną próbą zmiany typu. W zmiennej F2 typu FIGURE zapisany jest obiekt typu KOLO. Próba zmiany typu zmiennej F2 na KWADRAT musi się skończyć błędem, gdyż do zmiennych typu KWADRAT nie wolno przypisywać obiektów typu KOLO. Analogicznie, w zmiennej F1 zapisany jest obiekt typu KWADRAT. Próba zmiany typu zmiennej F1 na KOLO musi się skończyć błędem, gdyż do zmiennych typu KOLO nie wolno przypisywać obiektów typu KWADRAT.



W języku SQL można również wykorzystać operator TREAT do uzyskania dostępu do atrybutów charakterystycznych dla podtypów. Zapytanie wykorzystujące operator TREAT przedstawiono na przykładzie (1). W zapytaniu tym, wyświetlany jest KOLOR figur odczytywanych z tabeli FIGURY, oraz wartości atrybutów DLUGOSC_BOKU i PROMIEN specyficznych dla typów KWADRAT i KOLO. Użycie operatora TREAT jest tutaj analogiczne do jego użycia w programie w PL/SQL, przy czym zamiast zmiennej podano tutaj wywołanie operatora VALUE, który zwraca obiekt odczytany z tabeli. Typem wyrażenia VALUE(X) jest typ FIGURA, gdyż reprezentuje on obiekty odczytywane z tabeli obiektowej FIGURY. Za pomocą operatora TREAT typ ten jest zmieniany na typ podany w zapytaniu (KWADRAT albo KOLO). Po zmianie typu możliwe jest odczytanie wartości specyficznych dla typów KWADRAT i KOLO. Operator TREAT w języku SQL jednak nie zgłasza błędu, jeżeli zmiana typu nie jest możliwa, ale zwraca wartość NULL. Odwołanie się do atrybutu za pomocą operatora kropkowego, poprzez zmienną o wartości NULL również zwraca NULL. Dzięki temu, próba odczytania wartości atrybutu, który w obiekcie danego podtypu nie istnieje, kończy się odczytaniem wartości NULL. Jeżeli spojrzeć na wynik zapytania (1) można łatwo zauważyć, że dla niebieskiej figury (czyli koła) odczytano długość promienia, a długość boku jest równa NULL, a dla figury zielonej (czyli kwadratu) odczytano długość boku, a promień ma wartość NULL.



Slajd pokazuje hierarchię typów obiektowych, która zostanie użyta do przedstawienia kolejnego zagadnienia z modelu obiektowo – relacyjnego oraz później do zadań. Korzeniem hierarchii typów jest typ ISTOTA z którego dziedziczą podtypy KREGOWIEC i BEZKREGOWIEC. KREGOWIEC posiada podtypy SSAK i GAD, a BEZKREGOWIEC posiada podtypy MIECZAK i STAWONOG. Każdy z tych typów posiada atrybuty charakterystyczne dla siebie, jak również odziedziczone z nadtypów. Skrypt, który tworzy pokazaną na rysunku hierarchię typów został załączony do kursu (gatunki.sql). Skrypt ten tworzy również tabelę obiektową ISTOTY, do której następnie zapisuje obiekty każdego z typów przedstawionych na slajdzie.



Ostatnią rzeczą jaka zostanie omówiona na zajęciach z obiektowo-relacyjnych baz danych jest operator IS OF. Operator ten potrafi sprawdzić jakiego faktycznie typu jest obiekt. Istnieją dwie wersje operatora IS OF:

1) zmienna IS OF (typ obiektowy)

2) zmienna IS OF (ONLY typ obiektowy)

Pierwsza wersja sprawdza, czy obiekt zapisany w zmiennej jest typu podanego jako parametr albo dowolnego podtypu tego typu i jeżeli tak jest, to zwraca wartość logiczną TRUE, a w przeciwnym wypadku zwraca FALSE. Druga wersja sprawdza, czy obiekt zapisany w zmiennej jest dokładnie tego typu co podany jako parametr.

Rozważmy przykładowe zapytanie przedstawione na slajdzie. Zapytanie (1) odszukuje nazwy gatunków zapisane we wszystkich obiektach składowanych w tabeli obiektowej ISTOTY, które dodatkowo spełniają pewien warunek. Warunek (2) sprawdza, czy obiekt jest typu KREGOWIEC za pomocą pierwszej wersji operatora. Zapytanie wykorzystujące ten operator zwróci zatem gatunki wszystkich KREGOWCOW, jak również SSAKOW i GADOW, które dziedziczą z typu KREGOWIEC. Warunek (3) sprawdza, czy obiekt jest typu KREGOWIEC za pomocą drugiej wersji operatora. Zapytanie wykorzystujące ten operator zwróci zatem jedynie gatunki zapisane w obiektach typu KREGOWIEC, ale pominie SSAKI i GADY.

Istnieje również wersja tego operatora, która zwraca zaprzeczone wyniki: IS NOT OF (...).

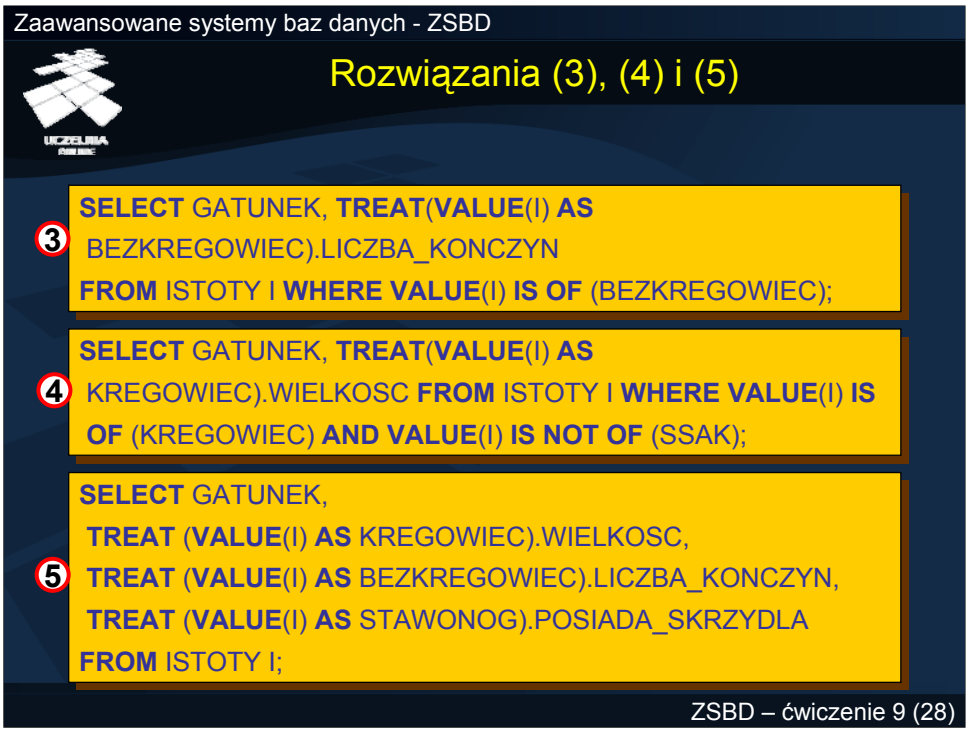
Zadania

3. Odnajdź nazwy gatunku i liczby kończyn wszystkich bezkręgowców.
4. Wypisz nazwy gatunków i rozmiary wszystkich kręgowców, które nie są ssakami. Nie korzystaj z operatorów zbiorowych.
5. Wypisz gatunki wszystkich zwierząt, a także, dla każdego z tych zwierząt: wielkość, liczbę_kończyn oraz czy posiada skrzydła. Jeżeli dla danego zwierzęcia, informacja ta się nie odnosi, to w odpowiedniej komórce tabeli wynikowej ma być wartość null.

(!) Zadania wykonaj za pomocą zapytań SQL do tabeli obiektowej ISTOTY.

Zadania – cd.

6. Odnajdź wszystkie ssaki i wypisz skojarzone z nimi dane (kolor futra, wielkość, gatunek).
7. Wypisz wszystkie bezkręgowce, które nie są ani mięczakami, ani stawonogami. Nie wykorzystuj operatorów zbiorowych.
8. Wypisz wszystkie istoty, które są kręgowcami i bezkręgowcami. Nie wykorzystuj operatorów zbiorowych.
9. Odnajdź wszystkie bezkręgowce, które mają więcej niż 6 kończyn.

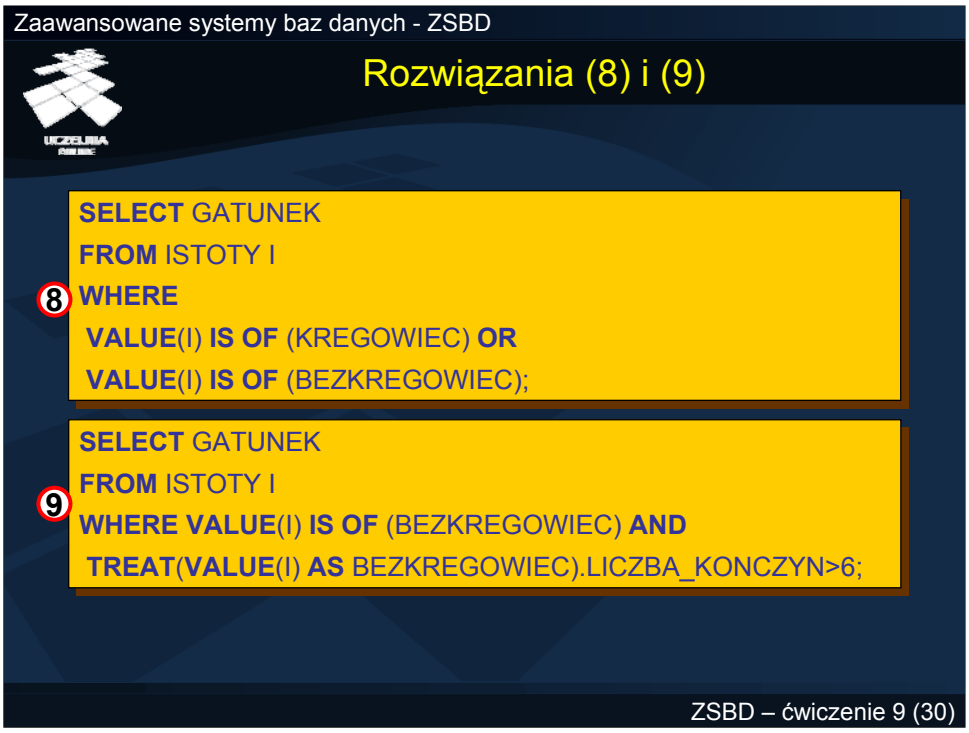


Slajd pokazuje rozwiązania zadań (3) (4) i (5), których treść przytoczono poniżej.

- (3)Odnajdź nazwy gatunku i liczby kończyn wszystkich bezkręgowców.
- (4)Wypisz nazwy gatunków i rozmiary wszystkich kręgowców, które nie są ssakami. Nie korzystaj z operatorów zbiorowych.
- (5)Wypisz gatunki wszystkich zwierząt, a także, dla każdego z tych zwierząt: wielkość, liczbe_kończyn oraz czy posiada skrzydła. Jeżeli dla danego zwierzęcia, informacja ta się nie odnosi, to w odpowiedniej komórce tabeli wynikowej ma być wartość null.

Slajd pokazuje rozwiązania zadań (6) i (7), których treść przytoczono poniżej.

- (6)Odnajdź wszystkie ssaki i wypisz skojarzone z nimi dane (kolor futra, wielkość, gatunek).
- (7)Wypisz wszystkie bezkręgowce, które nie są ani mięczakami, ani stawonogami. Nie wykorzystuj operatorów zbiorowych.



Slajd pokazuje rozwiązania zadań (8) i (9), których treść przytoczono poniżej.

- (8)Wypisz wszystkie istoty, które są kręgowcami i bezkręgowcami. Nie wykorzystuj operatorów zbiorowych.
- (9)Odnajdź wszystkie bezkręgowce, które mają więcej niż 6 kończyn.

Podsumowanie (1)

- W trakcie zajęć poznaliście Państwo sposób tworzenia hierarchii typów obiektowych.
- Dowiedzieliście się jak można stworzyć typ dziedziczący z innego typu, jak przesłonić odziedziczone metody, jak stworzyć metody i typy abstrakcyjne, jak można zmienić typ obiektu zapisanego w zmiennej nadtypu, oraz jak stwierdzić jaki jest faktyczny typ obiektu zapisanego w zmiennej.

Podsumowanie (2)

- Na zajęciach z obiektowo-relacyjnych baz danych nie przedstawiono tematyki związanej z perspektywami obiektowymi, ani tematyki związanej z odczytywaniem obiektów i kolekcji z poziomu obiektowego języka programowania. Zainteresowanych odsyłamy do dokumentacji SZBD Oracle:
  - Oracle® Database Concepts (Object Datatypes and Object Views)
  - Oracle® Database JDBC Developer's Guide and Reference (Working with Oracle Object Types)