JĘZYKI PROGRAMOWANIA CZĘŚĆ II

Paradygmat programowania obiektowego

dr inż. Barbara Fryc
bfryc@wsiz.edu.pl

PROGRAMOWANIE OBIEKTOWE

Programowanie obiektowe zakłada stworzenie stanów, czyli obiektów, które posiadają swoje dane oraz zachowań, czyli procesów lub metod. To, co odróżnia je od tradycyjnego proceduralnego programowania to fakt, że dane i procesy w programowaniu obiektowym nie tworzą ze sobą bezpośrednio tak ścisłego związku.

Podstawowymi pojęciami związanymi z programowaniem obiektowym są:

1. **Klasa**, która stanowi częściowy lub całościowy opis budowy i działania obiektów.
2. **Obiekty**, czyli fizyczne reprezentacje klasy, które zawierają dane oraz metody, które wykonują na nich określone zadania. Każdy obiekt dodatkowo posiadać trzy cechy:
   - **tożsamość** — cecha pozwalająca na odróżnienie go od innych obiektów;
   - **stan** — dane obiektu;
   - **zachowanie** — zestaw metod (funkcji) wykonujących zadania na danych.

KLASY W JAVIE

- Klasa w Javie jest typem danych i definicja klasy jest sposobem zdefiniowania nowego typu danych.
- Klasy tworzymy za pomocą słowa kluczowego **class**. Ogólna struktura definicji klasy wygląda następująco:

```
class NazwaKlasy {
    // Wnętrze klasy:
    // Tutaj znajdują się definicje pól danych, metod
    // i klas wewnętrznych klasy
}
```

- Konwencja nazewnictwa przyjęta w Javie zakłada stosowanie w nazwach klas górnej notacji wielbłąda (UpperCamelCase).

POLA DANYCH

- Definicja pola danych jest definicją zmiennej. Cechą wyróżniającą pole danych od innych zmiennych jest to, że pole danych musi się znaleźć w obszarze definicji klasy.
- Ogólna składnia definicji pola danych:

  NazwaTypu nazwaPola;

- Np:

  int predkoscSamochodu;

- W przypadku nazw pól danych w Javie stosuje się dolną notację wielbłąda (lowerCamelCase).

METODY

- Metoda jest funkcją należącą do definicji klasy.
- Ogólna składnia definicji funkcji:

```
TypRezultatu nazwaMetody(ListaParametrówFormalnych){
    //treść metody
}
```

- Np:

```
int podajPredkoscSamochodu(){
    return predkoscSamochodu;
}
```

- W przypadku metod konwencja nazewnictwa Javy zakłada stosowanie dolnej notacji wielbłąda.

DEFINICJA KLASY

- Poniżej znajduje się przykładowa definicja klasy Samochod z polem danych i metodą zdefiniowanymi wcześniej:

```
class Samochod{
    int predkoscSamochodu;
    int podajPredkoscSamochodu(){
            return predkoscSamochodu;
    }
}
```

TWORZENIE OBIEKTÓW – OPERATOR *NEW*

- Jak wcześniej powiedziano, definicja klasy jest jedynie wzorem, według którego będą tworzone obiekty. Dopiero mając konkretne obiekty możemy wykonywać w programie konkretne działania.
- Obiekty w Javie mogą być tworzone tylko w sposób dynamiczny (na stercie, w C++ można je tworzyć również na stosie).
- Do tworzenia obiektów służy operator *new*, po którym umieszcza się wywołanie konstruktora klasy. Ogólna składnia:

  ```
  new KonstruktorKlasy(ArgumentyKonstruktora);
  ```

- Np. dla klasy Samochod:

  ```
  new Samochod();
  ```

- Nazwa konstruktora jest tożsama z nazwą klasy, a więc w przypadku klasy Samochod, również konstruktor będzie miał tę samą nazwę.

REFERENCJE

- Obiekty tworzone na stercie są dostępne tylko poprzez bezpośrednie odwołania do pamięci (w języku C++ służą do tego wskaźniki).
- W języku Java nie ma wskaźników. Zamiast tego są **referencje**.
- Cechą odróżniającą referencję od wskaźnika jest to, że referencji nie można dowolnie przypisywać adresów, tak jak to ma miejsce w przypadku wskaźników. Referencje mogą wskazywać na to co zostanie zwrócone przez operator new lub na wartość null.
- Dzięki temu referencje są znacznie bezpieczniejsze od wskaźników.
- Definicja referencji wygląda jak definicja zmiennej:

  TypObiektu nazwaReferencji;
- Np:

  Samochod mojSamochod;

REFERENCJE

- Mając zdefiniowaną referencję możemy ją wykorzystać do wskazania na utworzony obiekt:

  ```
  Samochod mojSamochod;      //definicja referencji
  mojSamochod = new Samochod();      //przypisanie adresu
  ```

- Możemy także zainicjalizować referencję bezpośrednio w momencie tworzenia:

  ```
  Samochod mojSamochod = new Samochod();
  ```

- Referencji, która ma nie wskazywać na żaden obiekt możemy przypisać jawnie wartość null:

  ```
  Samochod mojSamochod = null;
  ```

- Jeśli referencja jest polem danych, to zabieg ten jest zbędny gdyż zostanie ona domyślnie zainicjalizowana wartością null.

REFERENCJE

- Na dany obiekt może wskazywać kilka referencji. Ustawienie referencji na konkretny obiekt odbywa się za pośrednictwem operatora przypisania, np:

```
Samochod mojSamochod1;
Samochod mojSamochod2;
mojSamochod1 = new Samochod();
mojSamochod2 = new Samochod();
mojSamochod1 = mojSamochod2;
```

- W powyższym przykładzie zdefiniowane zostały 2 referencje, do tych referencji przypisane zostały dwa osobne obiekty, a na końcu jedna referencja została przypisana do drugiej. W efekcie obydwie referencja wskazują na obiekt utworzony jako drugi, a an obiekt utworzony jako pierwszy nie wskazuje żadna z referencji.

WYWOŁYWANIE METOD

- Mając referencję do obiektu możemy wykonywać na nim wszelkie dostępne operacje.
- Najczęściej wykonywaną operacją jest wywołanie metody na rzecz obiektu.
- Wywołanie metody odbywa się z wykorzystaniem notacji kropkowej:

  referencja.nazwaMetody();
- Np:

  mojSamochod.podajPredkosc();
- Cechą odróżniającą wywołanie metody od wywołania funkcji jest to, że metoda jest wywoływana na rzecz obiektu i operacje przez nią wykonywane dotyczą tego właśnie obiektu. Funkcję wywołuje się na rzecz całego programu.

OPERACJE NA POLACH DANYCH

- Referencje mogą zostać także wykorzystane do operacji na polach danych. Robi się to w sposób analogiczny jak przy wywoływaniu metod, czyli poprzez notację kropkową:
- Ogólna składnia:

  referencja.poleDanych
- Przypisanie wartości polu danych będzie wyglądało następująco:

  mojSamochod.predkoscSamochodu = 100;
- Odczyt wartości z pola danych i wypisanie tej wartości na ekranie wygląda następująco:

  System.out.println(mojSamochod.predkoscSamochodu);
- Opisane tutaj operacje będą raczej rzadko występowały w profesjonalnych programach. Bezpośredni dostęp do pól danych jest niezgodny z zasadami enkapsulacji (stan obiektu ma być dostępny poprzez metody).

ENKAPSULACJA

- Jak widać na rysunku, metody stanowią rodzaj płaszcza ochronnego (kapsuły) uniemożliwiającej bezpośredni dostęp do zmiennych. Jest to **interfejs obiektu** (to co jest widoczne na zewnątrz)
- Zaletą stosowania enkapsulacji jest ułatwienie użytkownikowi posługiwania się obiektem oraz zabezpieczenie tego obiektu przed nieumyślnym uszkodzeniem (metodę można tak napisać, aby zmieniała stan obiektu w sposób dla niego bezpieczny).
- Enkapsulacja znacznie ułatwia **modularyzację** programów.
- Kody o dobrej strukturze modularnej mogą być wielokrotnie wykorzystane.
- Modularyzacja ma szczególne znaczenie w przypadku przemysłowej produkcji oprogramowania, gdzie pracują zespoły wieloosobowe w długich przedziałach czasowych.

KONSTRUKTORY

- **Konstruktory** są kolejnym elementem programu, który możemy umieszczać w definicji klasy.
- Jak sama nazwa wskazuje, konstruktor jest narzędziem tworzenia obiektów danej klasy (najczęściej konstruktor inicjalizuje stan obiektu, czyli przypisuje wartości polom danych).
- Konstruktor jest bardzo podobny do metody, tzn. posiada listę argumentów i ciało, czyli zbiór instrukcji wykonywany podczas wywołania konstruktora.
- Nazwa konstruktora jest zawsze identyczna z nazwą klasy, dla której został on zdefiniowany.
- W przeciwieństwie do metod, konstruktory nie zwracają żadnej wartości (nie określamy typu zwracanego w definicji konstruktora).
- Konstruktora nie wywołuje się na rzecz istniejącego obiektu. Jest on wywoływany w momencie tworzenia obiektu, wraz z operatorem new.

KONSTRUKTORY

- Ogólna składnia definicji konstruktora:

```
NazwaKonstruktora(lista argumentów formalnych){
    //zbiór instrukcji wykonywanych przez konstruktor
}
```

- W przykładowej klasie Samochod, konstruktor może przyjąć następującą postać:

```
class Samochod{
    int predkoscSamochodu;

    Samochod(int predkoscPoczatkowa){
            predkoscSamochodu = predkoscPoczatkowa;
    }

}
```

KONSTRUKTORY

- Mając zdefiniowany konstruktor dla klasy Samochod, mamy możliwość utworzenia obiektu w następujący sposób:

  ```
  new Samochod(20);
  ```

- Każda klasa, którą definiujemy bez konstruktora jest wyposażona w domyślny konstruktor bezargumentowy. W przypadku zdefiniowania własnego konstruktora klasy domyślny konstruktor staje się niedostępny. W takiej sytuacji operacja, która wcześniej była poprawna:

  ```
  new Samochod();
  ```

- teraz jest już błędem kompilacji.
- Jeśli chcemy, aby klasa miała do dyspozycji konstruktor bezargumentowy, to należy go zdefiniować w sposób jawny.

KONSTRUKTORY

- Klasa z dwoma konstruktorami będzie wyglądała następująco:

```
class Samochod{
    int predkoscSamochodu;

    Samochod(int predkoscPoczatkowa){
            predkoscSamochodu = predkoscPoczatkowa;
    }

    Samochod(){ }
}
```

- Powyższa sytuacja nosi nazwę **przeciążenia konstruktora**. Jest to przypadek analogiczny do przeciążenia metody. O tym który konstruktor zostanie wywołany zdecyduje lista argumentów.

SŁOWO KLUCZOWE *THIS*

- Słowo kluczowe **this** identyfikuje obiekt, w którym zostało użyte (jest referencją do tego obiektu, czyli sposobem na to, aby obiekt miał referencję do samego siebie).
- Słowo to pozwala odnosić się do pól danych i metod obiektu, a także umieszczać w ciele konstruktora wywołania innych konstruktorów.
- Ściślej rzecz ujmując, słowo this dodawane jest w sposób niejawny przed wszystkimi polami danych i wywołaniami metod, które odnoszą się do obiektu.

SŁOWO KLUCZOWE *THIS*

- this może być użyteczne w sytuacji przesłonięcia pola danych klasy, np:

```
class Klasa{
    int a;
    int b=this.a;
    void metoda(int a){
            this.a=a;
    }
}
```

- W powyższym przykładzie argument metody ma taką samą nazwę jak pole danych i przesłania je. W takiej sytuacji słowo this wskazuje, że chodzi o pole danych obiektu, a nie argument metody.
- W przypadku inicjalizacji jednego pola danych drugim, słowo this jest zbędne, ale nie jest błędem.

SŁOWO KLUCZOWE *THIS*

- Słowo this można także wykorzystać do wywołania konstruktora.
- Wywołanie takie może się znaleźć tylko w innym konstruktorze tej samej klasy (przeciążenie konstruktora).
- Przykład:

```
class A{
    int a;
    int b;
    A(int a){
            this.a = a;
    }
    A(int a, int b){
            this(a);     //to wywołanie musi być zawsze na pierwszym miejscu
            this.b = b;
    }
}
```

USUWANIE OBIEKTÓW Z PAMIĘCI – MECHANIZM ZBIERANIA ŚMIECI

- Przykład:

```
{
    {
            NazwaKlasy ob=new NazwaKlasy();
            //tutaj obiekt jest dostępny
    }
    //tutaj obiekt już nie jest dostępny (przeznaczony do usunięcia)
}
```

- Referencja została zdefiniowana jako zmienna lokalna w wewnętrznym zakresie. Po wyjściu z tego zakresu referencja przestaje istnieć, a więc na obiekt nie wskazuje już żadna referencja.
- Zbieracza śmieci można uruchomić jawnie poprzez wywołanie: `System.gc();`

FINALIZOWANIE OBIEKTÓW – METODA *FINALIZE()*

- Język Java nie posiada destruktorów (element języka C++).
- Istnieją sytuacje, w których wymagane jest wykonanie pewnych operacji przed usunięciem obiektu z pamięci (najczęściej chodzi tu o operacje czyszczące i uwolnienie wykorzystywanych zasobów, jak np. połączenie z bazą danych).
- Operacje, które należy wykonać bezpośrednio przed usunięciem z pamięci umieszcza się w metodzie finalize()
- Metodę tę należy zdefiniować w tworzonej przez nas klasie wg. wzoru:

```
protected void finalize() {
    //tutaj kod wykonywany bezpośrednio przed usunięciem obiektu
}
```

TYP WYLICZENIOWY - ENUM

- Tworzenie enumeratora (typu wyliczeniowego) nie różni się za bardzo od tworzenia klasy, wystarczy zamienić słowo **class** na **enum**:

```
public enum Rozmiar {
    XS, S, M, L, XL;
}
```

- Podobnie jak w przypadku normalnych klas tak i w przypadku typów wyliczeniowych może on posiadać atrybuty czy metody. Możesz także stworzyć klasę wyliczeniową, która będzie miała swój własny konstruktor inny od domyślnego.
- Do wartości typu wyliczeniowego odnosimy się jak do pól statycznych klasy. Dzieję się tak ponieważ w rzeczywistości wartości typu wyliczeniowego mają **automatycznie dodane modyfikatory public static final**.

```java
enum Color {
    1 usage
    RED( r: 255,  g: 0,  b: 0),    GREEN( r: 0,  g: 255, b: 0),  BLUE( r: 0,  g: 0,  b: 255);
    3 usages
    public int r, g, b;

    3 usages
    Color(int r, int g, int b)
    {
        this.r=r;
        this.g=g;
        this.b=b;
    }
    1 usage
    void chandeR(int r)
    {
        this.r=r;
    }
}
public class Main {
    public static void main(String[] args)  {
        Color c1 = Color.RED;
        Color c2 = Color.GREEN;
        c1.chandeR( r: 222);
        System.out.println(c1.r);
    }
}
```

DZIEDZICZENIE – PODSTAWOWE INFORMACJE

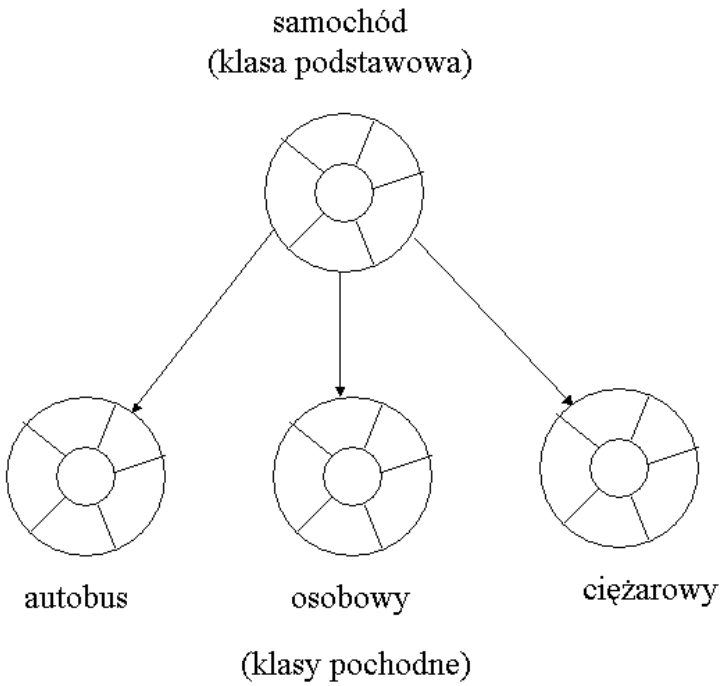
- **Dziedziczenie** jest jednym z podstawowych elementów języka orientowanego obiektowo.
- Jedną z cech charakteryzujących mechanizm dziedziczenia jest możliwość ponownego wykorzystania elementów zdefiniowanej wcześniej klasy.
- Dzięki takiemu rozwiązania redukujemy rozmiar programu, oraz czas programisty, który byłby potrzebny na wielokrotne zdefiniowanie tych samych elementów programu.
- Dziedziczenie opiera się na stworzeniu **struktury hierarchicznej**.
- Na szczycie tej struktury są klasy mające charakterystykę najbardziej ogólną.
- W miarę przesuwania się w dół hierarchii dziedziczenia przesuwamy się w kierunku typów bardziej szczegółowych.

DZIEDZICZENIE – PODSTAWOWE INFORMACJE

- Klasa znajdująca się na górze hierarchii dziedziczenia nosi nazwę **klasy podstawowej** (lub rodzica).
- Klasa, która dziedziczy po innej klasie nosi nazwę **klasy pochodnej** (lub potomka).
- Klasa potomna przejmuje wszystkie składniki klasy podstawowej, które ta udostępnia.
- Dodatkowo klasa potomna daje możliwość zdefiniowania własnych składników. W ten sposób klasa potomna **rozszerza funkcjonalność** klasy podstawowej.
- Oprócz tego klasa potomna może **przedefiniować** funkcjonalność odziedziczoną po klasie podstawowej.

PRZYKŁAD DZIEDZICZENIA

- W poniższym przykładzie trzy klasy dziedziczą po jednej klasie podstawowej.
- Typ *Samochód* jest tutaj typem bardziej ogólnym, niż np. typ Autobus.



DZIEDZICZENIE – PODSTAWOWE INFORMACJE

- Gdybyśmy nie stosowali dziedziczenia, to w każdej klasie musielibyśmy osobno definiować elementy, które są wspólne dla wszystkich typów (np. różnych typów samochodów).
- Innym rozwiązaniem byłoby zdefiniowanie jednej dużej klasy, która zawiera wszelką możliwą funkcjonalność (np. wszelkie możliwe metody jakie możemy wykorzystać dla różnych typów samochodów). Podstawową wadą takiego rozwiązania jest fakt, że klasa skonstruowana w ten sposób udostępniałaby funkcjonalność, która jest w danej chwili niepotrzebna (np. w samochodzie osobowym mielibyśmy funkcje typowe dla samochodu ciężarowego).

DZIEDZICZENIE – PODSTAWOWE INFORMACJE

- Klasa dziedzicząca po innej klasie może być także klasą, po której mogą dziedziczyć inne klasy.
- W konsekwencji dziedziczenie pozwala na zbudowanie z klas **wielopoziomowej** struktury drzewiastej.
- Klasy w kolejnych poziomach dziedziczą składniki po klasie, której są bezpośrednimi potomkami, jak również po wszystkich innych klasach które znajdują się wyżej w danej gałęzi dziedziczenia.

MODELE DZIEDZICZENIA

- Poszczególne języki programowania różnią się miedzy sobą w szczegółowym podejściu do modelu dziedziczenia.
- **Jednokrotne dziedziczenie** zakłada, że dana klasa może mieć tylko jedną klasę podstawową. W przypadku **wielokrotnego dziedziczenia** dana klasa może mieć więcej niż jedną klasę podstawową.
- Przykładem języka, w którym obowiązuje jednokrotne dziedziczenie jest język Java, natomiast wielokrotne dziedziczenie jest możliwe w języku C++.

MODELE DZIEDZICZENIA

- Innym aspektem odróżniającym języki jest sposób budowy hierarchii dziedziczenia.
- Niektóre języki zakładają, że wszystkie klasy mają **wspólny korzeń** hierarchii dziedziczenia, czyli istnieje taka klasa, po której dziedziczą wszystkie inne.
- W innym podejściu programista może tworzyć wiele **odrębnych hierarchii** dziedziczenia.
- Przykładem języka, który ma wspólny korzeń hierarchii dziedziczenia jest język Java. Wspólnym rodzicem klas zdefiniowanych w Javie jest klasa java.lang.Object.
- Przykładem języka, w którym tworzy się wiele niezależnych hierarchii dziedziczenia jest język C++.

DZIEDZICZENIE W JAVIE

- Aby zadeklarować dziedziczenie w języku Java, wykorzystuje się słowo kluczowe extends.
- Ogólna składnia definicji klasy dziedziczącej po innej klase:

```
class NazwaKlasy extends NazwaNadklasy {
        // Ciało klasy – tutaj znajduje się to co w normalnej klasie
}
```

- Jeśli nie zadeklarujemy jawnie dziedziczenia, to nie oznacza to, że dziedziczenia nie ma. W takiej sytuacji klasa dziedziczy bezpośrednio po wspólnym przodku wszystkich klas, czyli klasie java.lang.Object.

PRZYKŁADOWA HIERARCHIA DZIEDZICZENIA

- Przykładowa definicja klasy podstawowej może wyglądać następująco:

```
class SrodekTransportu{
    int liczbaPasazerow;
    float ladownosc;
    int getLiczbaPasazerow (){
            return liczbaPasazerow);
    }
    void setLiczbaPasazerow (int liczbaPasazerow){
            this.liczbaPasazerow=liczbaPasazerow;
    }
    float getLadownosc (){
            return ladownosc;
    }
    void setLadownosc (int ladownosc){
            this.ladownosc=ladownosc;
    }
}
```

PRZYKŁADOWA HIERARCHIA DZIEDZICZENIA

- Klasa dziedzicząca po SrodekTransportu może rozszerzyć klasę podstawową poprzez następującą definicję:

```
class LadowySrodekTransportu extends SrodekTransportu{
    byte liczbaKol;
    byte getLiczbaKol(){
            get liczbaKol);     }
    void setLiczbaKol(byte liczbaKol){
            this.liczbaKol=liczbaKol;  }
}
```

- Mając tak zdefiniowaną hierarchię dziedziczenia możemy utworzyć zarówno obiekty klasy podstawowej, jak i potomnej:

```
SrodekTransportu st = new SrodekTransportu();
LadowySrodekTransportu lst = new
LadowySrodekTransportu();
```

PRZYKŁADOWA HIERARCHIA DZIEDZICZENIA

- Na rzecz obiektu st możemy wywołać wszystkie metody zdefiniowane w klasie SrodekTansportu oraz odwołać się do wszystkich jego pól danych, np:

  st.getLadownosc();

  st.liczbaPasazerow=5;

- W przypadku obiektu lst możemy wywołać wszystkie metody zdefiniowane w klasie LadowySrodekTransportu oraz uzyskać dostęp do wszystkich jego pól danych, np:

  lst.getLiczbaKol();

  lst.liczbaKol=4;

PRZYKŁADOWA HIERARCHIA DZIEDZICZENIA

- Dodatkową możliwością, jaką daje nam dziedziczenie jest możliwość odwołania się do składników klasy podstawowej poprzez referencję do obiektu klasy potomnej. Oznacza to, że poprawne będą operacje:

  lst.getLadownosc();

  lst.liczbaPasazerow=4;

- Obiekt klasy potomnej pozwala wykorzystywać wszystkie składniki, które odziedziczył po klasie podstawowej. Z punktu widzenia użytkownika obiektu nie ma rozróżnienia pomiędzy tymi dwoma rodzajami składników.
- Powyższe rozumowanie jest poprawne w przypadku, gdy obydwie klasy należą do tego samego pakietu. W ogólnym przypadku klasa podstawowa i potomna mogą należeć do różnych pakietów. Wówczas konieczne jest zastosowanie odpowiednich **modyfikatorów dostępu**.

PRZYKŁADOWA HIERARCHIA DZIEDZICZENIA

- Jak zostało powiedziane wcześniej, hierarchia dziedziczenia może mieć strukturę wielopoziomową.
- Struktura dziedziczenia złożona z klas SrodekTransportu oraz LadowySrodekTransportu składa się tak na prawdę z trzech klas (podstawową jest klasa Object).
- Oznacza to, że obydwie klasy, które zdefiniowaliśmy odziedziczyły składniki klasy Object.
- W konsekwencji możemy wykorzystać zarówno obiekt st, jak i lst, do wywołania składników klasy Object, np:
  - st.clone(); – metoda służąca do kopiowania obiektu.
  - lst.toString(); – metoda zamieniająca obiekt na postać łańcucha znakowego,
  - lst.equals(innyObiekt) – porównanie obiektu lst z innym obiektem.

PRZYKŁADOWA HIERARCHIA DZIEDZICZENIA

- Przedstawiony przykład hierarchii dziedziczenia można rozbudować o następny poziom:

```
class Samochod extends LadowySrodekTransportu{
    String rodzajSilnika;
    String getRodzajSilnika(){
            return rodzajSilnika);
    }
    void setRodzajSilnika(String rodzajSilnika){
            this.rodzajSilnika=rodzajSilnika;
    }
}
```

PRZYKŁADOWA HIERARCHIA DZIEDZICZENIA

- Zdefiniowana klasa dziedziczy składniki wszystkich klas znajdujących się wyżej w gałęzi dziedziczenia.
- Oznacza to, że poprawne będą operacje:

```
Samochod s = new Samochod();
s.clone();
s.setLiczbaPasazerow(10);
s.getLiczbaKol();
s.getRodzajSilnika();
```

MODYFIKATORY

- Zarówno klasy, jak i składniki klas mogą posiadać szereg modyfikatorów. Modyfikatory te dzielimy na dwie kategorie:
  - **modyfikatory dostępu** – określają widoczność danego elementu programu w innych częściach programu,
  - **modyfikatory właściwości** – określają szczególne właściwości danego elementu programu.
- Do modyfikatorów dostępu zaliczamy:
  - public – stosowany do klas i ich składników, oznacza dostępność z każdego miejsca w programie,
  - protected – stosowany tylko do składników klas, oznacza widoczność w obszarze klasy, jej klas potomnych oraz pakietu,
  - package – modyfikator domyślny (nie jest pisany jawnie), stosowany do klas i ich składników, oznacza widoczność tylko w obszarze pakietu,
  - private – stosowany tylko do składników klas, oznacza widoczność tylko w obszarze klasy.

MODYFIKATORY

- Poniższa tabela zawiera zbiorczą informację o modyfikatorach dostępu:

Modyfikator	*klasa*	*pakiet*	*podklasa*	*wszędzie*
private	X			
package	X	X		
protected	X	X	X	
public	X	X	X	X

ZASŁANIANIE NAZW

- Przy dziedziczeniu może wystąpić sytuacja, w której klasa posiada pola danych lub metody o identycznych nazwach jak jej nadklasa.
- W takiej sytuacji składniki klasy dziedziczącej **przesłaniają** składniki klasy dziedziczonej.
- W przypadku metod, aby zaistniało przesłonięcie, muszą jeszcze być takie same listy argumentów. W przeciwnym przypadku mielibyśmy do czynienia z **przeładowaniem** metody.
- Przeładowanie metod ma miejsce zarówno w przypadku metod znajdujących się w tej samej klasie jak i metod znajdujących się na różnych poziomach hierarchii dziedziczenia.

ZASŁANIANIE NAZW

- Przesłonięcie zilustrujemy na przykładzie. Załóżmy, że mamy dwie klasy powiązane relacją dziedziczenia:

```
public class Rodzic {
    public int i=0;
    public void metoda(){
            System.out.println("To jest metoda z klasy Rodzic " +i);
    }
}
public class Potomek extends Rodzic {
}
```

- Tutaj mamy do czynienia ze zwykłym dziedziczeniem. Operacje:

```
Potomek p = new Potomek();
p.metoda();
```

- spowodują wywołanie metody z klasy Rodzic.

ZASŁANIANIE NAZW

- Zmodyfikujemy teraz klasę Potomek w następujący sposób:

```
public class Potomek extends Rodzic {
    public int i=1;   //zmienna przesłaniająca
    public void metoda(){ //metoda przesłaniająca
            System.out.println("To jest metoda z klasy Potomek "+i);
    }
}
```

- Zawartość klasy Potomek przesłania zawartość klasy Rodzic. Oznacza to, że operacje:

```
Potomek p = new Potomek();
p.metoda();
p.i=2;
```

- spowodują wywołanie metody z klasy Potomek. Dodatkowo wartość pola tej klasy zostanie zmodyfikowana (pole klasy Rodzic pozostanie niezmienione).

PRZEŁADOWANIE METOD

- Możemy teraz dodać do klasy Rodzic nową metodę:

```
public class Rodzic {
    public int i=0;
    public void metoda(){
            System.out.println("To jest metoda z klasy Rodzic " +i);
    }
    public void metoda(int i){
            System.out.println("To jest metoda przeładowująca z klasy Rodzic
    " +i);
    }
}
```

- Metoda, która została dodana nie jest przesłaniana przez zawartość klasy Potomek.

PRZEŁADOWANIE METOD

- Mając klasę Rodzic z przeładowaną metodą możemy wykonać następujące operacje:

```
Rodzic r = new Rodzic();
r.metoda();
r.metoda(3);
Potomek p = new Potomek();
p.metoda();
p.metoda(4);
```

- Jak widać przeładowanie jest dostępne zarówno z poziomu klasy podstawowej jak i potomnej dzięki dziedziczeniu.
- Gdybyśmy metodę przeładowującą zdefiniowali w klasie potomnej zamiast w klasie podstawowej, to wówczas byłaby ona dostępna tylko dla obiektów klasy potomnej.

SŁOWO KLUCZOWE SUPER

- Składniki przesłonięte nie są dostępne z zewnątrz obiektu. Oznacza to, że mając obiekt danej klasy, np:

```
Potomek p = new Potomek();
```

- nie ma sposobu aby poprzez referencję p uzyskać dostęp do przesłoniętych składników klasy Rodzic.
- Dostęp do składników przesłoniętych jest jednak możliwy z wnętrza klasy potomnej. Do tego celu wykorzystuje się słowo kluczowe super.
- Słowo super pozwala wywoływać przesłonięte metody, konstruktory z nadklasy, a także uzyskiwać dostęp do przesłoniętych pól danych.

SŁOWO KLUCZOWE SUPER

- Aby zilustrować działanie słowa super uzupełnimy metodę w klasie Potomek o odpowiednie instrukcje:

```
public class Potomek extends Rodzic {
    public int i=1;   //zmienna przesłaniająca
    public void metoda(){ //metoda przesłaniająca
            super.metoda();  //przesłonięta metoda
            super.i=4;  //przesłonięte pole danych
            System.out.println("To jest metoda z klasy Potomek "+i);
    }
}
```

- W przedstawionym przykładzie wywołując metodę przesłaniającą powodujemy jednocześnie wywołanie metody przesłoniętej. Dodatkowo modyfikowane jest przesłonięte pole danych.

SŁOWO KLUCZOWE SUPER

- Wykorzystanie słowa super do wywołania konstruktorów przydaje się w przypadkach, kiedy mamy do czynienia z przeładowanymi konstruktorami w klasie nadrzędnej.
- Aby to zilustrować zdefiniujemy w klasie Rodzic dwa konstruktory:

```
public class Rodzic {
    public int i;
    public Rodzic() {}
    public Rodzic(int i) { ths.i = i;}
}
```

- Należy pamiętać, że pierwszą operacją jaką wykonuje konstruktor jest wywołanie innego konstruktora (z nadklasy lub z tej samej klasy). Jeśli nie podaliśmy jawnie jaki to ma być konstruktor to będzie wywoływany konstruktor bezargumentowy z nadklasy.

SŁOWO KLUCZOWE SUPER

- Wskazanie konstruktora z nadklasy do wywołania odbywa się w sposób analogiczny, jak w przypadku słowa this w odniesieniu do konstruktorów z tej samej klasy.
- W klasie potomnej zdefiniujemy konstruktor wywołujący jawnie konstruktor z nadklasy:

```
public class Potomek extends Rodzic {
    public int i=1;   //zmienna przesłaniająca
    public Potomek(int i){
            super(i);
            this.i=i;
    }
}
```

- Zdefiniowany konstruktor inicjalizuje zarówno pole klasy potomnej jaki pole rodzica (to drugie za pośrednictwem konstruktora rodzica)

SŁOWO KLUCZOWE SUPER

- Należy pamiętać, że wywołanie konstruktora przez super musi się znaleźć na pierwszej pozycji w kodzie konstruktora klasy potomnej. Automatycznie eliminuje to możliwość wykorzystania słowa this do wywołania konstruktora tej samej klasy.
- Należy także pamiętać, że w przypadku zdefiniowania konstruktora w klasie, domyślny bezargumentowy konstruktor staje się niedostępny. Jeśli klasa podstawowa nie będzie miała takiego konstruktora to może się okazać, że w konstruktorach klas potomnych zgłaszany jest błąd kompilacji. Wynika on z tego, że konstruktory te próbują wywołać konstruktor domyślny, którego nie ma. Aby tego uniknąć zazwyczaj definiuje się dla klas konstruktory bezargumentowe, nawet jeśli nie wykonują one żadnych specjalnych zadań.

METODY WIRTUALNE I POLIMORFIZM

- Polimorfizm oznacza wielopostaciowość i odnosi się do metod. W przypadku dziedziczenia pojęcie polimorfizmu nabiera szczególnego znaczenia.
- Wiadomo, że zmiennych referencyjnych możemy używać do wskazywania na obiekty typu zgodnego z typem referencji.
- Obiekt zgodny z typem referencji to jest przede wszystkim obiekt tej samej klasy, ale także obiekty wszystkich klas potomnych.
- Dla przykładu rozważmy hierarchię dziedziczenia: FiguraGeometryczna->Trojkat->TrojkatRownoboczny. Możemy powiedzieć że trójkąt jest w szczególności figurą geometryczną. Tak samo trójkąt równoboczny jest w szczególności przykładem trójkąta, ale także jest przykładem figury geometrycznej. W konsekwencji stwierdzamy **wszystkie typy są jednocześnie swoimi typami nadrzędnymi**, gdyż dziedziczą po nich całą charakterystykę.

METODY WIRTUALNE I POLIMORFIZM

- Konsekwencją zgodności typów jest możliwość wskazywania referencją do typu podstawowego na obiekty klas potomnych.
- Poprawne będą więc operacje przypisania obiektów do referencji:

```
FiguraGeometryczna f1 = new FiguraGeometryczna();
FiguraGeometryczna t1 = new Trojkat();
FIguraGeometryczna tr1 = new TrojkatRownoboczny();
Trojkat t2 = new Trojkat();
Trojkat tr2 = new TrojkatRownoboczny();
```

- W szczególności pamiętając, że klasa java.lang.Object jest korzeniem wszystkich hierarchii dziedziczenia, stwierdzamy, że referencja tego typu może być użyta do wskazywania na obiekty dowolnego typu.

METODY WIRTUALNE I POLIMORFIZM

- Załóżmy teraz, że w klasach Trojkat i TrojkatRownoboczny mamy do czynienia z przesłonięciem metody o nazwie obliczPowierzchnie().
- Rozważmy następujące operacje:

```
Trojkat t = new TrojkatRownoboczny();
t.obliczPowierzchnie();
```

- Pytanie brzmi: Która metoda została wywołana?
- Odpowiedź: **O tym która metoda zostaje wywołana decyduje typ obiektu, a nie typ referencji, która na niego wskazuje.**
- Należy także zauważyć, że wywołanie metody nie byłoby możliwe, gdyby nie była ona zdefiniowana w klasie Trojkat. Okazuje się, że mając daną referencję możemy wywoływać różne metody w zależności od typu obiektu, na który wskazuje referencja. Metody takie noszą nazwę **metod wirtualnych**.

METODY WIRTUALNE I POLIMORFIZM

- W szczególności daną referencję możemy wykorzystać do pokazywania na obiekty różnych typów potomnych:

```
Trojkat t = new TrojkatRownoboczny();
t.obliczPowierzchnie();
t = new TrojkatRownoramienny();
t.obliczPowierzchnie();
t = new TrojkatProstokatny();
t.obliczPowierzchnie();
```

- W powyższym przykładzie wywołanie metody wygląda identycznie w każdym przypadku, jednak za każdym razem jest to wywołanie innej metody.

METODY WIRTUALNE I POLIMORFIZM

- Metody wirtualne to takie, które mogą mieć różną postać w zależności od typu obiektu. W języku Java **wszystkie metody są wirtualne**.
- O metodzie, która przesłania metodę z klasy podstawowej mówimy, że **przedefiniowuje** (ang. override) metodę z klasy podstawowej.
- Nie wszystkie obiektowe języki programowania definiują wszystkie metody jako wirtualne. W językach takich jak C++ lub C# należy zadeklarować metodę jako wirtualną. W przeciwnym przypadku będziemy mieli do czynienia nie z przedefiniowaniem metody ale jej ukryciem. W takiej sytuacji o tym, która metoda zostanie wywołana zdecyduje typ referencji, a nie typ obiektu.

KOMPOZYCJA

- Kompozycja pełni podobną funkcję, jaką pełni dziedziczenie. Oba te narzędzia pozwalają dodać nową funkcjonalność do obiektu.
- Działanie kompozycji polega na umieszczaniu **obiektów jako składników innych obiektów**.
- Kompozycję stosuje się wtedy, gdy chcemy mieć funkcjonalność istniejącej klasy wewnątrz definiowanej przez siebie klasy bez dodawania do klasy nowych elementów interfejsu (taka sytuacja ma miejsce w przypadku dziedziczenia).
- W praktyce technika kompozycji jest znacznie częściej stosowana w programowaniu obiektowym niż technika dziedziczenia.

KOMPOZYCJA

- Aby zilustrować działanie kompozycji rozważymy przykład klasy Samochod. Wiadomo, że obiekt Samochod jest obiektem złożonym z wielu elementów składowych (silnik, układ hamulcowy, układ elektryczny, układ kierowniczy, itp.).
- Rozważając zastosowanie dziedziczenia lub kompozycji w danym przypadku powinniśmy rozważyć relację jaka zachodzi pomiędzy obiektami. W przypadku dziedziczenia powinna to być relacja "jest przykładem" lub "jest szczególnym przypadkiem". W przypadku kompozycji powinna to być relacja "jest składnikiem".
- W przypadku samochodu żaden z jego elementów składowych nie jest w relacji typowej dla dziedziczenia z obiektem samochodu (np. silnik nie jest szczególnym przypadkiem samochodu). Powinniśmy więc zastosować kompozycję.

KOMPOZYCJA

- Definicja klasy Samochod z wykorzystaniem kompozycji będzie wyglądała następująco:

```
class Samochod{
    Silnik silnik = new Silnik();
    UkladChlodzacy ukladChlodzacy = new UkladChlodzacy();
    UkladHamulcowy ukladHamulcowy = new UkladHamulcowy();
}
```

- Mając utworzony obiekt tej klasy możemy wykonywać operacje na jego poszczególnych elementach składowych (np. wywoływać metody):

```
Samochod s = new Samochod();
s.silnik.zwiekszObroty(2000);
```

KOMPOZYCJA

- Stosując kompozycję powinniśmy pamiętać o enkapsulacji (nieupublicznianie pól danych klasy):

```
public class Samochod{
    private Silnik silnik = new Silnik();
    public void zwiekszObroty(int obroty) {
            silnik.zwiekszObroty(obroty);
    }
}
```

- Wtedy operacja zwiększenia obrotów będzie wyglądała następująco:

```
Samochod s = new Samochod();
s.zwiekszObroty(2000);
```

KOMPOZYCJA

- Możemy także zastosować metodę dostępową (getter) do pobrania obiektu składowego:

```
public class Samochod{
    private Silnik silnik = new Silnik();
    public Silnik getSilnik() {
            return silnik;
    }
}
```

- Wtedy operacja zwiększenia obrotów będzie wyglądała następująco:

```
Samochod s = new Samochod();
s.getSilnik().zwiekszObroty(2000);
```

KOMPOZYCJA

- Operacja zwracania referencji do obiektu składowego jest wprawdzie poprawna, lecz zawiera błąd metodologiczny.
- Należy pamiętać, że w języku Java zarówno przesyłanie argumentów obiektowych do metod, jak i ich zwracanie odbywa się **poprzez referencję** (nie przez wartość). Oznacza to, że nie ma tworzenia kopii obiektu a przesyłany jest jedynie jego adres (tak jak w przypadku wskaźników w C++).
- Metoda zwracająca referencję do wewnętrznego obiektu klasy psuje enkapsulację (udostępnia na zewnątrz obiekt prywatny).
- W sytuacji tego typu metoda powinna tworzyć kopię obiektu składowego i zwracać do niego referencję.
- Kopiowanie obiektu wykonuje się najczęściej poprzez mechanizm klonowania (przedefiniowanie metody clone()).

KOMPOZYCJA VS. DZIEDZICZENIE

- Podsumujemy teraz najważniejsze cechy kompozycji i dziedziczenia
- Kompozycja:
  - jest implementowana w prosty sposób poprzez przesyłanie wszystkich wywołań do pól danych klasy,
  - nie wprowadza zależności od szczegółów implementacji obiektu będącego polem danych,
  - posiada większą elastyczność ze względu na tworzenie dynamiczne (w czasie wykonania programu), a nie w czasie kompilacji.
- Dziedziczenie :
  - niszczy enkapsulację, gdyż implementacja pod- i nadklasy są ze sobą ściśle związane,
  - nowe metody dodane do nadklasy zmieniają funkcjonalność podklasy,
  - pod- i nadklasa muszą być rozwijane równolegle,
  - zaprojektowanie klasy, którą można łatwo rozszerzać wymaga większego nakładu pracy.

ABSTRAKCJE – PODSTAWOWE INFORMACJE

- Abstrakcja w programowaniu polega na zdefiniowaniu pewnej funkcjonalności bez jej implementowania.
- Sposób wprowadzania abstrakcji zależy od konkretnego języka programowania. Język Java posiada dwa narzędzia służące definiowaniu abstrakcji: **klasy abstrakcyjne** i **interfejsy**.
- Zarówno klasy abstrakcyjne jak i interfejsy są sposobem zdefiniowania pewnego typu danych (podobnie jak klasa).
- Kluczowe znaczenie dla definiowania abstrakcji mają **metody abstrakcyjne**. Są to metody, które definiują samą sygnaturę metody bez jej implementacji.
- Metody abstrakcyjne mogą być definiowane tylko w klasach abstrakcyjnych lub interfejsach.

KLASY ABSTRAKCYJNE – PODSTAWOWE INFORMACJE

- **Klasy abstrakcyjne** posiadają charakterystykę podobną do zwykłych klas. Zasadniczą cechą wyróżniającą klasy abstrakcyjne jest brak możliwości utworzenia obiektu klasy abstrakcyjnej.
- W skład klasy abstrakcyjnej może wchodzić wszystko to co może wchodzić w skład zwykłej klasy. Dodatkowo w klasach abstrakcyjnych mamy możliwość definiowania metod abstrakcyjnych (zwykła klasa nie może zawierać takich metod).
- Definicję klasy abstrakcyjnej oznaczamy modyfikatorem abstract:

```
abstract class NazwaKlasy{
   //wszystko to co może być w zwykłej klasie +
   //definicje metod abstrakcyjnych;
}
```

METODY ABSTRAKCYJNE – PODSTAWOWE INFORMACJE

- **Metody abstrakcyjne** definiuje się podając ich sygnaturę (typ zwracany, nazwa, lista argumentów).
- Dodatkowo metodę abstrakcyjną oznaczamy modyfikatorem abstract (w przypadku braku tego modyfikatora treść metody nie może zostać pominięta).
- Ogólna składnia definicji metody abstrakcyjnej:

```
abstract TypZwracany nazwaMetody(argumenty);
```

- Niemożliwość utworzenia obiektu klasy abstrakcyjnej wynika z istnienia metod abstrakcyjnych. Utworzenie obiektu klasy abstrakcyjnej powodowałoby powstanie obiektu o niezdefiniowanych zachowaniach.
- Metody abstrakcyjne muszą zostać zaimplementowane w klasach potomnych klasy abstrakcyjnej (o ile klasy potomne nie są klasami abstrakcyjnymi).

PRZYKŁAD KLASY ABSTRAKCYJNEJ

- Definiowanie klasy abstrakcyjnej zilustrujemy na przykładzie klasy Pocztowka.
- Klasa będzie posiadała składniki wspólne dla wszystkich rodzajów pocztówek:

```
abstract class Pocztowka
{
   String adresat;
   abstract void tresc();
}
```

- Metoda tresc() została zdefiniowana jako metoda abstrakcyjna. Metody abstrakcyjne definiujemy wtedy, gdy nie jesteśmy w stanie lub nie chcemy zdefiniować konkretnej implementacji.

PRZYKŁAD KLASY ABSTRAKCYJNEJ

- Istnienie klasy abstrakcyjnej ma uzasadnienie wtedy, kiedy jest ona dziedziczona przez klasę nieabstrakcyjną, w której zdefiniujemy konkretne implementacje metod abstrakcyjnych.
- W przypadku dziedziczenia klasy abstrakcyjnej przez klasę nieabstrakcyjną implementacja metod abstrakcyjnych będzie wymogiem poprawności kompilacji. Umieszczanie w klasie metod abstrakcyjnych jest więc sposobem na wymuszenie na programiście zdefiniowania pewnych zachowań klasy.
- Jeśli klasą potomną klasy abstrakcyjnej jest również klasa abstrakcyjna, to wówczas implementacja metod nie jest wymagana.
- Dla klasy Pocztowka zdefiniujemy trzy klasy potomne: PocztowkaImieninowa, PocztowkaWielkanocna, PocztowkaNoworoczna.

PRZYKŁAD KLASY ABSTRAKCYJNEJ

- Klasa PocztowkaImieninowa:

```
class PocztowkaImieninowa extends Pocztowka{
    PocztowkaImieninowa( String a )  {    adresat = a;  }
    void tresc()  {
            System.out.println("Adresat: "+a);
            System.out.println("Najlepsze życzenia imieninowe!");
    }
}
```

- Implementacja metody abstrakcyjnej nie różni się niczym od zwykłej definicji metody. Jedyny wymóg to zgodność z sygnaturą metody abstrakcyjnej.

PRZYKŁAD KLASY ABSTRAKCYJNEJ

```
class PocztowkaWielkanocna extends Pocztowka{
    PocztowkaWielkanocna ( String a ){
            adresat = a;
    }
    void tresc(){
            System.out.println("Adresat: "+adresat);
            System.out.println("Wesolego Alleluja!");
    }
}
class PocztowkaNoworoczna extends Pocztowka {
    int rok;
    PocztowkaNoworoczna ( String a, int r ){
            adresat = a;
            rok     = r;
    }
    void tresc(){
            System.out.println("Adresat: "+adresat);
            System.out.println("Szczesliwego Nowego Roku "+rok);
    }
}
```

POLIMORFIZM METOD ABSTRAKCYJNYCH

- W przypadku metod abstrakcyjnych polimorfizm jest szczególnie ważny. Takie metody definiuje się właśnie po to, aby je przedefiniować w klasach potomnych. Inaczej ich istnienie nie miałoby sensu.
- Obowiązują takie same zasady jak w przypadku zwykłych klas. Nie możemy wprawdzie stworzyć obiektu klasy abstrakcyjnej, ale możemy definiować referencje i wskazywać nimi na obiekty klas potomnych. Oznacza to, że poprawne będą operacje:

```
Pocztowka p = new PocztowkaNoworoczna( "Kasia", 2004 ) ;
p.tresc();
p = new PocztowkaImieninowa( "Tomek" ) ;
p.tresc();
p = new PocztowkaWielkanocna( "Romek" ) ;
p.tresc();
```

KONWERSJA TYPU REFERENCJI

- Referencja typu podstawowego wskazująca na obiekty klas potomnych pozwala uzyskać dostęp tylko do składników klasy potomnej, które zostały odziedziczone po klasie podstawowej. Oznacza to, że np. niepoprawna będzie następująca operacja:

```
Pocztowka p = new PocztowkaNoworoczna( "Kasia", 2004 ) ;
p.rok=2008;
```

- Wynika to z faktu, że w typie Pocztowka nie ma informacji o polu rok.
- Dostęp do takich pól danych można uzyskać stosując rzutowanie typu referencji na typ potomny:

```
((PocztowkaNoworoczna)p).rok=2008;
```

- Wszystkie użyte tu nawiasy są niezbędne, gdyż pozwalają narzucić odpowiednią kolejność operacji.

KONWERSJA TYPU REFERENCJI

- Podobnej konwersji musimy dokonywać przy podstawieniu jednego typu do drugiego.
- Załóżmy, że mamy dwa obiekty wskazywane przez dwie referencje (przyjmujemy, że Pocztowka nie jest klasą abstrakcyjną):

```
Pocztowka p=new Pocztowka(…);
PocztowkaNoworoczna n=new PocztowkaNoworoczna(…);
```

- Przypisanie referencji typu potomnego do referencji typu podstawowego (p=n;) nie stanowi problemu.
- Niemożliwa natomiast jest operacja w przeciwną stronę (n=p;). Wynika to z faktu, że referencja n jest typu rozszerzającego typ p. Może się więc zdarzyć, że referencja będzie posiadała informację o składnikach, których nie ma w obiekcie, na który wskazuje.

KONWERSJA TYPU REFERENCJI

- Rozwiązaniem przedstawionego problemu jest rzutowanie typów:

n=(PocztowkaNoworoczna)p;

- Operacja taka uniemożliwia odwołanie się poprzez referencję n do składników, które nie zostały zdefiniowane w klasie Pocztowka.
- Generalnie możliwe są konwersje na linii rodzic-potomek.
- Bezpośrednie konwersje pomiędzy typami na tym samym poziomie hierarchii dziedziczenia (np. pomiędzy klasą PocztowkaImieninowa, a PocztowkaNoworoczna) nie są możliwe.
- Można takie konwersje zrealizować poprzez rzutowanie etapami (najpierw na wspólny typ podstawowy, a później na jego typ potomny).

INTERFEJSY

- Interfejsy są podobnie jak klasy sposobem zdefiniowania typu w języku Java.
- Interfejsy definiują typy czysto abstrakcyjne. Oznacza to, że metody, które zdefiniujemy w interfejsie mogą być tylko metodami abstrakcyjnymi (w klasach abstrakcyjnych możemy definiować zarówno zwykłe metody, jak i metody abstrakcyjne).
- Obok metod w skład interfejsu mogą wchodzić pola danych typów prymitywnych. Pola te muszą być finalne i statyczne.
- W interfejsach możemy umieszczać także interfejsy wewnętrzne (zagnieżdżone).
- Wszystkie składniki interfejsu są publiczne.
- Podobnie jak w przypadku klas abstrakcyjnych, interfejsy w sposób bezpośredni nie mogą być wykorzystywane do tworzenia obiektów.

INTERFEJSY

- Interfejsy definiuje się z wykorzystaniem słowa kluczowego interface:

```
interface NazwaInterfejsu{
    //definicje stałych statycznych, metod abstrakcyjnych
    //oraz interfejsów zagnieżdżonych
}
```

- Przykład interfejsu:

```
interface IPrzykladInterfejsu{
    int poleInterfejsu=10;
    int obliczWartosc(int a);
}
```

- Nie jest to ogólnie przyjęta konwencja ale niektórzy przyjmują zasadę rozpoczynania nazwy klasy od dużej litery I. Pozostałe zasady nazewnictwa są takie same jak dla klas.

INTERFEJSY

- Interfejsy możemy definiować z dostępem publicznym lub domyślnym (obszar pakietu).
- Składniki interfejsu mogą mieć tylko dostęp publiczny. Nie ma konieczności używania modyfikatorów dostępu w definicji interfejsu, gdyż zawsze przypisywany jest modyfikator public.
- Ponadto nie musimy stosować modyfikatorów final, static i abstract, gdyż są to modyfikatory stosowane zawsze do metod w interfejsach.
- W konsekwencji poprzednia definicja jest równoważna następującej:

```
interface IPrzykladInterfejsu{
    public final static int poleInterfejsu=10;
    public abstract int obliczWartosc(int a);
}
```

- Pole danych zawsze musi mieć przypisaną wartość w definicji interfejsu.

MODYFIKATOR FINAL W ZASTOSOWANIU DO ZMIENNYCH

- Modyfikator final w zastosowaniu do pól danych i zmiennych lokalnych powoduje, że wartość która jest przechowywana nie może zostać zmieniona (jest stała).
- Cechą odróżniającą modyfikator final od const z języka C++ jest brak wymogu inicjalizacji w momencie definicji.
- Zmienną finalną możemy zainicjalizować w miejscu jej definicji:

```
final int zmiennaFinalna= 0;
```

- Możemy także zainicjalizować ją później. Jeżeli jest to zmienna lokalna zdefiniowana np. w metodzie to można ją zainicjalizować dalej w kodzie tej metody:

```
final int zmiennaFinalna;
. . .
zmienaFinalna = 0;
```

MODYFIKATOR FINAL W ZASTOSOWANIU DO ZMIENNYCH

- Jeśli zdefiniowaliśmy finalne pole danych bez przypisania wartości, to operacja ta będzie wymagana w konstruktorze.
- Do momentu przypisania wartości zmiennej finalnej pozostaje ona tzw. czystą zmienną, której nie możemy użyć (próba użycia niezainicjalizowanej zmiennej będzie sygnalizowana jako błąd kompilacji).
- Zaletą braku wymogu przypisania wartości w przypadku pól danych jest możliwość przypisania innej wartości każdemu z obiektów danej klasy (przy modyfikatorze const w C++ takie coś jest niemożliwe).
- W przypadku interfejsów inicjalizacja pola danych jest bezwzględnie wymagana. Wynika to z faktu, że interfejs nie może posiadać kodu, w którym taka inicjalizacja mogłaby zostać wykonana (brak konstruktora).

Modyfikator final w zastosowaniu do klas i metod

- Modyfikator final może też być stosowany jako modyfikator metod oraz modyfikator klas.
- W przypadku metod powoduje on, że metoda nie może zostać przedefiniowana w klasach potomnych.
- W przypadku klas modyfikator powoduje, że klasa nie może być dziedziczona przez inne klasy (automatycznie wszystkie metody takiej klasy stają się finalne).

IMPLEMENTACJA INTERFEJSÓW

- Istnienie interfejsu w programie jest uzasadnione tylko wtedy, gdy istnieją klasy implementujące (dziedziczące) ten interfejs.
- Wskazanie implementacji interfejsu w definicji klasy odbywa się poprzez słowo kluczowe implements (analogicznie jak wskazanie dziedziczenia poprzez extends):

```
class NazwaKlasy implements NazwaInterfejsu1,
      NazwaInterfejsu2, ...
{     standardowe składniki klasy + implementacje metod
      abstrakcyjnych z interfejsów;
}
```

- Model dziedziczenia w Javie nie przewiduje wielokrotnego dziedziczenia (które jest możliwe np. w języku C++). Możliwa jest jednak implementacja wielu interfejsów, co ma zastąpić wielokrotne dziedziczenie klas.

IMPLEMENTACJA INTERFEJSÓW

- Implementacja interfejsu przez klasę nie wyklucza możliwości dziedziczenia.
- Definicja klasy jednocześnie dziedziczącej klasę (tylko jedną) i implementującej interfejsy będzie wyglądała następująco:

class NazwaKlasy extends KlasaPodstawowa *implements NazwaInterfejsu1, NazwaInterfejsu2, ...*

{ standardowe składniki klasy + implementacja metod abstrakcyjnych z interfejsów;

}

- Interfejs może być implementowany zarówno przez zwykłą klasę, jak i przez klasę abstrakcyjną. W przypadku implementacji przez zwykłą klasę wymagana jest implementacja wszystkich metod abstrakcyjnych interfejsu.

IMPLEMENTACJA INTERFEJSÓW

- Przykładowa implementacja interfejsu IPrzykladInterfejsu będzie wyglądała następująco:

```
class KlasaImplementujaca implements IPrzykladInterfejsu
{
    public int obliczWartosc(int a){
            return a*poleInterfejsu;
    }
}
```

- Przy implementacji interfejsu obowiązują normalne zasady dziedziczenia. W konsekwencji wszystkie pola danych z interfejsu są dostępne w klasie implementującej.

IMPLEMENTACJA INTERFEJSÓW

- Należy zwrócić uwagę, że modyfikator public przed implementacją metody z interfejsu jest jedynym możliwym. Wynika to z zasady, wg której dostęp do metody przedefiniowywanej lub implementowanej może być taki sam, jak w miejscu, z którego tę metodę dziedziczymy, lub szerszy (np. metoda z dostępem protected może mieć zostawiony taki sam dostęp lub zmieniony na public, ale nie może mieć zmienionego dostępu na private lub domyślny package).
- W przypadku interfejsów wszystkie metody są publiczne, wiec również w klasach implementujących musi być taki dostęp.

IMPLEMENTACJA INTERFEJSÓW

- Rozważmy następujący przykład interfejsu:

```
public interface Drapieznik {
    boolean polujNaOfiare(Ofiara p);
    void zjedzOfiare(Ofiara p);
}
```

- Powyższy interfejs może zostać zaimplementowany w następujący sposób:

```
public class Kot extends Ssak implements Drapieznik {
    public boolean polujNaOfiare(Ofiara p) {
            // tutaj specyficzny dla kota kod polowania na ofiarę
    }
    public void zjedzOfiare(Ofiara p) {
            // tutaj specyficzny dla kota kod jedzenia ofiary
    }
}
```

IMPLEMENTACJA INTERFEJSÓW

- Zakładamy teraz, że mamy interfejs

```
interface Ofiara{
    //tutaj składniki interfejsu
}
```

- Klasa implementująca:

```
public class Mysz extends Ssak implements Ofiara {
    // tutaj składniki klasy mysz + implementacje metod interfejsu
    Ofiara
}
```

IMPLEMENTACJA INTERFEJSÓW

- Mając klasy Kot i Mysz możemy wykonać następujące operacje:

```
Kot kot = new Kot();
Mysz mysz = new Mysz();
kot.polujNaOfiare(mysz);
kot.zjedzOfiare(mysz);
```

- Należy zwrócić uwagę, że przekazanie argumentu typu Mysz do metod wywoływanych na rzecz obiektu kot jest jak najbardziej poprawne. Wynika to z faktu, że obiekt, którego klasa implementuje interfejs jest zarówno typu swojej klasy ale też typu interfejsu (kot jest obiektem typu Kot oraz Drapieznik, mysz jest obiektem typu Mysz i Ofiara).

IMPLEMENTACJA INTERFEJSÓW

- W szczególności referencji typu interfejsu możemy używać do wskazywania na obiekty klas:

```
Drapieznik dKot = new Kot();
Ofiara oMysz = new Mysz();
```

- Referencje takie mają jednak pewne ograniczenia. Pozwalają one odwoływać się tylko do składników, które same posiadają. Oznacza to, że gdyby np. w klasie Kot znalazły się składniki, których nie ma w interfejsie Drapieznik to nie mogłyby one zostać wywołane. W takiej sytuacji należałoby stosować rzutowanie typów, np:

```
Drapieznik dKot = new Kot();
//błąd: dKot.metodaKota();
((Kot)dKot).metodaKota(); //dobrze
```

- Podobna sytuacja byłaby, gdyśmy poprzez referencję typu Ssak, wskazującą na obiekt typu Kot, próbowali wywołać metodę implementowaną z interfejsu Drapieznik.

DZIEDZICZENIE INTERFEJSÓW

- Interfejsy mogą dziedziczyć inne interfejsy. Nie ma tutaj ograniczeń co do liczby dziedziczonych interfejsów.
- Dziedziczenie interfejsów deklaruje się poprzez słowo kluczowe extends:

```
interface NazwaInterfejsu extends Interfejs1, Interfejs2, ...
{
   //tutaj normalne składniki interfejsu
}
```

- Interfejsy nie mogą dziedziczyć klas.

DZIEDZICZENIE INTERFEJSÓW

- Załóżmy, że mamy zdefiniowany wcześniej interfejs Drapieżnik oraz interfejs Jadowity. Możemy zdefiniować teraz nowy interfejs:

```
interface DrapieznikJadowity extends Drapieznik, Jadowity
{
    //tutaj normalne składniki interfejsu
}
```

- Przykładowa klasa implementująca:

```
class Grzechotnik extends Gad implements DrapieznikJadowity{
    //tutaj implementacje wszystkich metod z całej hierarchii
    interfejsów
}
```

- Daje to efekt podobny jak w przypadku implementacji każdego z interfejsów z osobna. Różnice występują tylko w przypadku wystąpienia w interfejsach pól danych o takiej samej nazwie. Możemy mieć wówczas do czynienia z przesłonięciem jednego pola przez drugie lub niejednoznacznością (błąd kompilacji).

STATYCZNE POLA DANYCH

- Statyczne pola danych deklaruje się poprzez umieszczenie modyfikatora static przed tym polem.
- Jeśli pole danych klasy jest deklarowane jako statyczne, to oznacza to, że istnieje dokładnie jedno wcielenie tego pola, niezależnie od liczby obiektów danej klasy utworzonych w programie.
- Pola statyczne istnieją nawet wtedy, gdy nie ma utworzonego żadnego obiektu danej klasy.
- Z punktu widzenia rezerwacji pamięci pola statyczne są podobne do zmiennych globalnych w języku C. Oznacza to, że pamięć zostaje przydzielana już w momencie załadowania programu do pamięci i pozostaje zarezerwowana przez cały czas działania programu.

STATYCZNE POLA DANYCH

- Statyczne pola danych nazywane są często **polami danych klasy**, w odróżnieniu od zwykłych pól danych, które są nazywane **polami danych obiektu**.
- Rozróżnienie to wynika z faktu, że statyczne pole danych nie jest związane z żadnym konkretnym obiektem klasy (jest współdzielone przez wszystkie obiekty tej klasy). W przypadku zwykłych pól każdy obiekt posiada swoją własną kopię tego pola i może przechowywać w nim inną wartość.
- Do pól statycznych klasy możemy odwoływać się zarówno za pośrednictwem referencji do obiektu jak i nazwy klasy (do zwykłych pól danych możemy się odwoływać tylko za pośrednictwem referencji).
- Zmiana wartości statycznego pola danych jest natychmiast widoczna we wszystkich obiektach tej klasy.

STATYCZNE POLA DANYCH

- Załóżmy, że mamy następującą klasę z definicją pola statycznego:

```
class A{
    static int poleStatyczne;
}
```

- Możemy wykonać teraz następujące operacje:

```
A.poleStatyczne=5;
System.out.println(A.poleStatyczne);
A ref = new A();
ref.poleStatyczne = 10;
System.out.println(A.poleStatyczne);
System.out.println(ref.poleStatyczne);
```

- Wyświetlona wartość nie zależy od tego, czy użyjemy nazwy klasy, czy też referencji.

STATYCZNE METODY

- Metodę statyczną definiuje się używając modyfikatora static.
- Metoda deklarowana jako statyczna, podobnie jak w przypadku pól danych jest nazywana **metodą klasy**.
- Metoda statyczna może być wywoływana za pośrednictwem nazwy klasy lub też referencji do obiektu określonej klasy.
- Oprócz sposobu wywoływania, metodę statyczną od niestatycznej różni to, że ta pierwsza ma dostęp jedynie do składników statycznych klasy. Metoda niestatyczna ma dostęp do jednych jak i drugich.
- Podobnie jak w przypadku pól danych metody niestatyczne nazywane są **metodami obiektu** i mogą być wywoływane jedynie za pomocą referencji do obiektu tej klasy (lub jednej z klas potomnych).
- Metody statyczne można przedefiniować jedynie metodami statycznymi.

STATYCZNE METODY

- Załóżmy, że mamy następującą klasę z polem statycznym i metodą statyczną:

```
class A{
    static int poleStatyczne;
    int poleNiestatyczne;
    static void metodaStatyczna(int a){
            poleStatyczne = a;
            //błąd: poleNiestatyczne = a;
    }
    void metodaNiestatyczna(int a){
            poleStatyczne = a;
            poleNiestatyczne = a;  //dobrze
    }
}
```

STATYCZNE METODY

- Mając zdefiniowaną klasę możemy wykonać następujące operacje:

```
A.metodaStatyczna(5);
//błąd: A.metodaNiestatyczna(10);
A ref = new A();
ref.metodaStatyczna(15);
ref.metodaNiestatyczna(15);
```

- Rezultat wywołania metody statycznej jest zawsze taki sam, niezależnie czy wywołano ją przez referencję, czy też poprzez nazwę klasy.

INICJALIZATOR STATYCZNYCH PÓL DANYCH

- Niekiedy istnieje potrzeba inicjalizacji statycznych pól danych, która nie jest tylko przypisaniem gotowej wartości, ale może mieć postać bardziej złożoną, np. może wymagać wykonania pewnej operacji w pętli.
- W takim przypadku nie możemy wykorzystać konstruktora, gdyż inicjalizacja musi być dokonana jeszcze przed utworzeniem obiektów danej klasy.
- W takiej sytuacji stosuje się **inicjalizator statycznych pól danych**.
- Operacje inicjalizujące umieszcza się w obszarze klasy, zamknięte w klamry poprzedzone słowem static:

```
static{
   //tutaj umieszczamy instrukcje inicjalizujące
}
```

INICJALIZATOR STATYCZNYCH PÓL DANYCH

- Przykład klasy z inicjalizatorem wykorzystanym do inicjalizacji tablicy:

```
class WektorStatyczny{
        static final int n = 10;
        static int tab[] = new int[n];
        static{
          for(int i=0;i<tab.length;i++)
                  tab[i]=1;
        }
}
```

- Rozmiar tablicy statycznej musi być statyczny.
- Wykonanie inicjalizacji nie wymaga dodatkowych czynności. Zostanie wykonane przy uruchomieniu programu zawierającego klasę.