# Relacja Products i Suppliers typu jeden do wielu

## Dodanie klasy Supplier

namespace GracjanFilipekEFLab

{

public class Supplier

{

public int SupplierID { get; set; }

public string CompanyName { get; set; }

public string City { get; set; }

}

}

## Dodanie pola reprezentującego dostawcę w klasie Product

Dodane pole jest nullifikowalne, ponieważ polecenie mówi, że najpierw należy dodać produkt, a potem dostawcę

public Supplier? SuppliedBy { get; set; }

## Dodanie Suppliers do kontekstu

public DbSet<Supplier> Suppliers { get; set;}

## Migracja bazy i jej aktualizacja

dotnet ef migrations add AddSuppliedByColumnInProducts

dotnet ef database update

W kolejnych zadaniach migracje i aktualizacje będą pomijane

## Dodanie produktu do bazy

ProductContext productContext = new ProductContext();

Console.WriteLine("Podaj nazwę produktu");

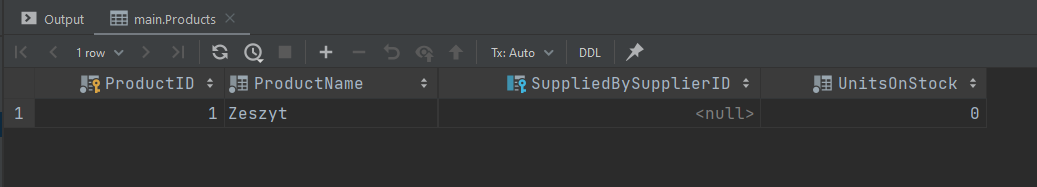
string prodName = Console.ReadLine();

Product product = new Product { ProductName = prodName };

productContext.Products.Add(product);

productContext.SaveChanges();

Tablica Products po wykonaniu:



## Dodanie dostawcy do Suppliers

ProductContext productContext = new ProductContext();

Console.WriteLine("Podaj nazwę dostawcy");

string companyName = Console.ReadLine();

Console.WriteLine("Podaj miasto");

string city = Console.ReadLine();

Supplier supplier = new Supplier { CompanyName = companyName, City = city};

productContext.Suppliers.Add(supplier);

productContext.SaveChanges();

## Powiązanie dostawcy i produktu

ProductContext productContext = new ProductContext();

var supplier = productContext.Suppliers.Single();

var query = from prod in productContext.Products

select prod;

foreach (var prod in query)

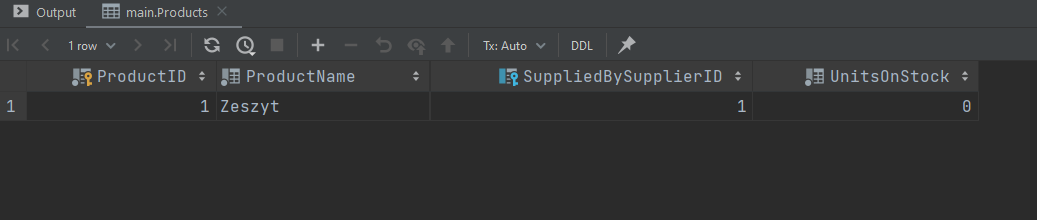
{

prod.SuppliedBy = supplier;

}

productContext.SaveChanges();

Tablica Products po wykonaniu:



# Odwrócenie relacji

## Modyfikacja klasy Product

namespace GracjanFilipekEFLab

{

public class Product

{

public int ProductID { get; set; }

public string ProductName { get; set; }

public int UnitsOnStock { get; set; }

}

}

## Modyfikacja klasy supplier

namespace GracjanFilipekEFLab

{

public class Supplier

{

public int SupplierID { get; set; }

public string CompanyName { get; set; }

public string City { get; set; }

public ICollection<Product> suppliedProducts { get; } = new HashSet<Product>();

}

}

## stworzenie produktów i dostawcy, powiązanie ich ze sobą i dodanie do bazy

ProductContext productContext = new ProductContext();

Supplier supplier = new Supplier { City = "London", CompanyName = "Gregory's Tea Shop"};

ICollection<Product> products = new List<Product>

{

new Product { ProductName = "Red tea", UnitsOnStock = 4},

new Product { ProductName = "Green tea", UnitsOnStock = 3 },

new Product { ProductName = "White tea", UnitsOnStock = 2 }

};

products.ForEach(p => { supplier.suppliedProducts.Add(p); });

productContext.Suppliers.Add(supplier);

productContext.Products.AddRange(products);

productContext.SaveChanges();

Tabela Suppliers po wykonaniu:

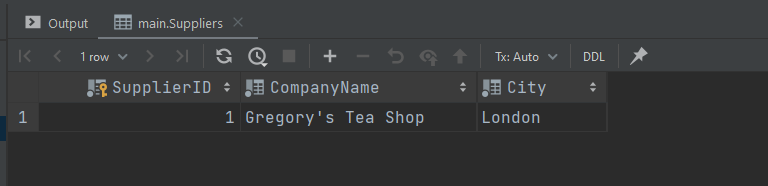
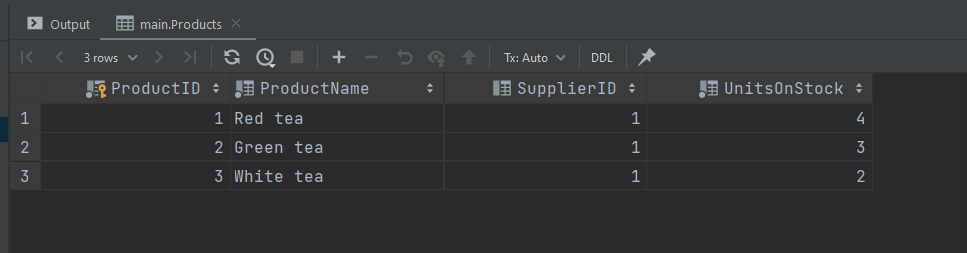


Tabela Products po wykonaniu:



# Relacja dwustronna

## Dodanie pola suppliedBy do klasy product

public Supplier SuppliedBy { get; set; }

## Dodanie pola suppliedProducts do klasy Supplier

Public ICollection<Product> suppliedProducts { get; } = new List<Product>();

## Stworzenie dostawcy, produktów i powiązanie ich

ProductContext productContext = new ProductContext();

Supplier supplier = new Supplier { City = "London", CompanyName = "Gregory's Tea Shop"};

productContext.Suppliers.Add(supplier);

ICollection<Product> products = new List<Product>

{

new Product { ProductName = "Red tea", UnitsOnStock = 4},

new Product { ProductName = "Green tea", UnitsOnStock = 3 },

new Product { ProductName = "White tea", UnitsOnStock = 2 }

};

productContext.Products.AddRange(products);

products.ForEach(p => {

p.SuppliedBy = supplier;

supplier.suppliedProducts.Add(p);

});

productContext.SaveChanges();

Tabela Suppliers po wykonaniu:

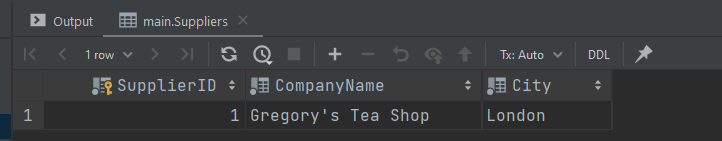
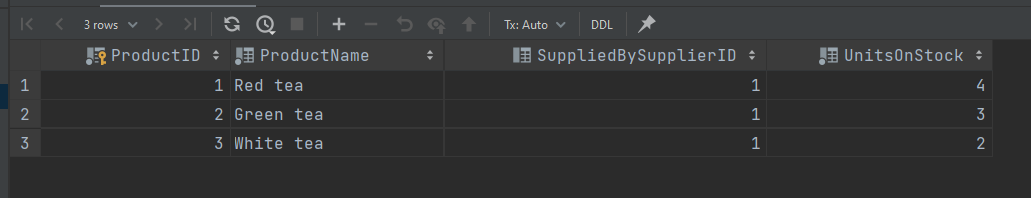


Tabela Products po wykonaniu:



## Komentarz

Na wstępie zaznaczam, że nie zrozumiałem, co miałoby oznaczać „Quantity” w klasie Invoice i dlaczego jest to cecha właśnie klasy Invoice (jak pokazuje diagram), a nie np. relacji „Includes”.

Z tego powodu na potrzeby tego zadania zakładam, że „Quantity” odnosi się do liczby sprzedanych sztuk danego produktu w ramach danej faktury. Z tego też powodu relację many-to-many zamodeluję jako dwie relacje one-to-many (tylko w taki sposób będę w stanie przypisać „Quantity” relacji faktura-produkt, a nie samej fakturze).

## klasA Invoice

public class Invoice

{

[Key]

public int InvoiceNumber { get; set; }

public ICollection<InvoiceProduct> InvoiceProducts { get; } = new HashSet<InvoiceProduct>();

}

## Klasa Product

public class Product

{

public int ProductId { get; set; }

[Required]

public string ProductName { get; set; }

[Required]

public int UnitsOnStock { get; set; }

public ICollection<InvoiceProduct> InvoiceProducts { get; } = new HashSet<InvoiceProduct>();

}

## Klasa InvoiceProduct

public class InvoiceProduct

{

public int InvoiceId { get; set; }

public Invoice Invoice { get; set; }

public int ProductId { get; set; }

public Product Product { get; set; }

[Required]

public int Quantity { get; set; }

public void Sell()

{

if (Product.UnitsOnStock - Quantity < 0)

{

throw new ArgumentOutOfRangeException("Cannot sell " + Quantity + " units, product has only " + Product.UnitsOnStock + " units available");

}

Product.UnitsOnStock -= Quantity;

Product.InvoiceProducts.Add(this);

Invoice.InvoiceProducts.Add(this);

}

}

## DbContext

public class ProductContext: DbContext

{

public DbSet<Product> Products { get; set;}

public DbSet<Invoice> Invoices { get; set;}

public DbSet<InvoiceProduct> InvoicesProduct { get; set;}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

base.OnConfiguring(optionsBuilder);

optionsBuilder.UseSqlite("Datasource=ProductsDatabase");

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<InvoiceProduct>().HasKey(p => new { p.InvoiceId, p.ProductId });

}

}

## Funkcja Main

static void Main(string[] args)

{

ProductContext productContext = new ProductContext();

ICollection<Invoice> invoices = new HashSet<Invoice>

{

new Invoice { },

new Invoice { },

new Invoice { },

};

ICollection<Product> products = new HashSet<Product>

{

new Product { ProductName = "Red tea", UnitsOnStock = 10},

new Product { ProductName = "Green tea", UnitsOnStock = 10 },

new Product { ProductName = "White tea", UnitsOnStock = 10 }

};

// connect invoices and products

ICollection<InvoiceProduct> invoiceProducts = new HashSet<InvoiceProduct>

{

new InvoiceProduct { Invoice = invoices.ElementAt(0), Product = products.ElementAt(0), Quantity = 1},

new InvoiceProduct { Invoice = invoices.ElementAt(0), Product = products.ElementAt(1), Quantity = 2},

new InvoiceProduct { Invoice = invoices.ElementAt(1), Product = products.ElementAt(2), Quantity = 3},

new InvoiceProduct { Invoice = invoices.ElementAt(2), Product = products.ElementAt(0), Quantity = 2},

new InvoiceProduct { Invoice = invoices.ElementAt(2), Product = products.ElementAt(1), Quantity = 1},

new InvoiceProduct { Invoice = invoices.ElementAt(2), Product = products.ElementAt(2), Quantity = 2},

};

// finalise each invoice

foreach (var invProd in invoiceProducts)

{

invProd.Sell();

}

productContext.Invoices.AddRange(invoices);

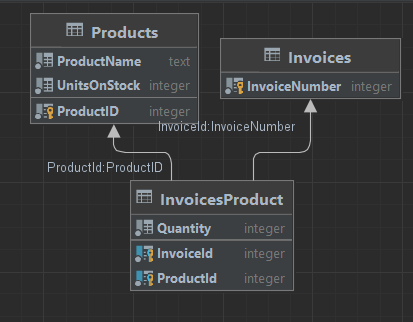
productContext.Products.AddRange(products);

productContext.SaveChanges();

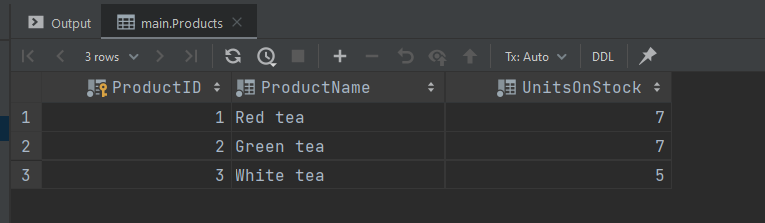
}

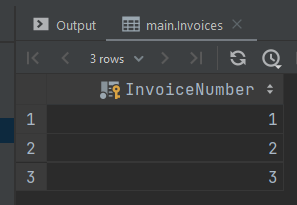
## Efekty

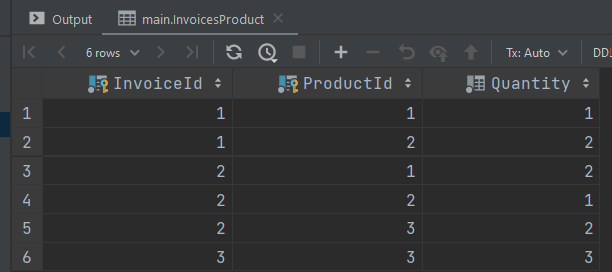
### Schemat bazy



### Zawartość tabeli







# Mapowanie dziedziczenia: Table-per-hierarchy

Polecenia zadań 5 i 6 nie są dla mnie jasne. Jedynymi “strategiami mapowania dziedziczenia”, o jakich znalazłem informacje są Table-Per-Hierarchy, Table-Per-Type oraz Table-Per-Concrete-Type. Polecenie zadania 5 podpunkt b mówi o trzech różnych strategiach mapowania (zakładam, że mowa o tych wyżej wymienionych), natomiast zadanie 6. każe zamodelować dziedziczenie strategią Table-Per-Type. Z tego powodu połączyłem oba te zadania i pokaże hierarchię zrealizowaną za pomocą tych trzech strategii w trzech kolejnych punktach.

## Kontekst

public DbSet<Company> Companies { get; set; }

public DbSet<Supplier> Suppliers { get; set;}

public DbSet<Customer> Customers { get; set; }

## Klasa Company

public class Company

{

[Key]

public string CompanyName { get; set; }

[Required]

public string Street { get; set; }

[Required]

public string City { get; set; }

[Required]

public string ZipCode { get; set; }

public override string ToString()

{

return JsonSerializer.Serialize(this);

}

}

## Klasa Customer

public class Customer: Company

{

public float Discount { get; set; } = 0;

## Klasa Supplier

public class Supplier: Company

{

[Required]

[MinLength(8)]

[MaxLength(20)]

public string BankAccountNumber { get; set; }

## klasa program

Klasa ta jest używana we wszystkich trzech kolejnych zadaniach

static void Main(string[] args)

{

//AddToDatabase();

//QuerryFromDatabase();

}

static void AddToDatabase()

{

ProductContext productContext = new ProductContext();

ICollection<Customer> customers = new HashSet<Customer>

{

new Customer { City = "London", CompanyName = "Customer1", Discount = 0.2f, Street = "St. James", ZipCode = "00-000"},

new Customer { City = "Paris", CompanyName = "Customer2", Discount = 0.0f, Street = "Trafalgar", ZipCode = "11-222"},

new Customer { City = "Budapest", CompanyName = "Customer3", Discount = 0.1f, Street = "Rouge st.", ZipCode = "33-444"},

};

ICollection<Supplier> suppliers = new HashSet<Supplier>

{

new Supplier { City = "Phoenix", CompanyName = "Supplier1", BankAccountNumber = "0123456789012345", Street = "St. Pierre", ZipCode = "55-6666"},

new Supplier { City = "Beijing", CompanyName = "Supplier2", BankAccountNumber = "3453453463242352", Street = "Tianan", ZipCode = "77-888"},

new Supplier { City = "Nan Madol", CompanyName = "Supplier3", BankAccountNumber = "4673529174837284", Street = "Uloka", ZipCode = "99-999"},

};

productContext.Customers.AddRange(customers);

productContext.Suppliers.AddRange(suppliers);

productContext.SaveChanges();

}

static void QuerryFromDatabase()

{

ProductContext productContext = new ProductContext();

var comp\_query = from comp in productContext.Companies

select comp;

Console.WriteLine("Companies:");

foreach (var company in comp\_query)

{

Console.WriteLine(company);

}

Console.WriteLine();

var supp\_query = from supp in productContext.Suppliers

select supp;

Console.WriteLine("Suppliers:");

foreach (var supplier in supp\_query)

{

Console.WriteLine(supplier);

}

Console.WriteLine();

var cust\_query = from cust in productContext.Customers

select cust;

Console.WriteLine("Customers");

foreach (var cust in cust\_query)

{

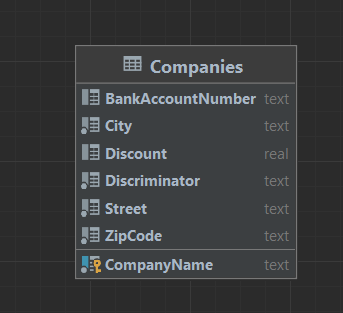
Console.WriteLine(cust);

}

}

## Efekt

Po wykonaniu funkcji main w bazie istnieje tylko jedna tablica:

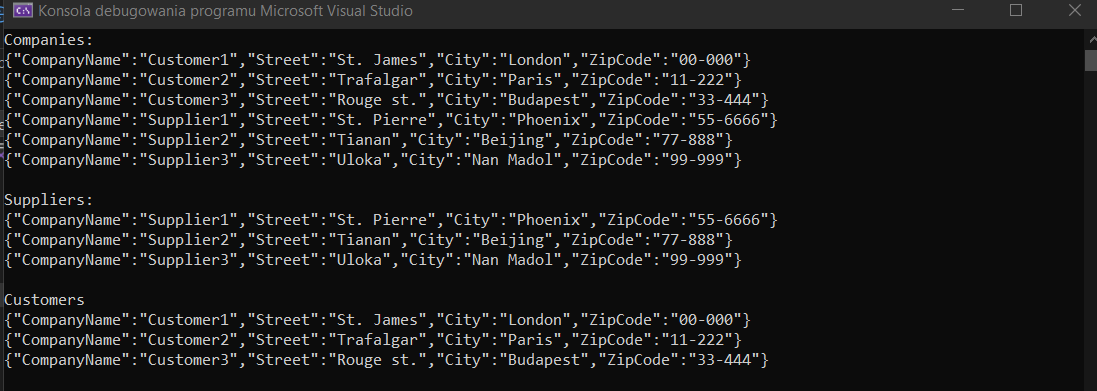


Zawartość tablicy:

Obraz zawierający tekst

Opis wygenerowany automatycznie

## Pobranie i wypisanie danych



# Mapowanie dziedziczenia: Table-per-type

## Kontekst

Kod kontekstu nie zmienił się względem poprzedniego podjeścia.

## Klasy

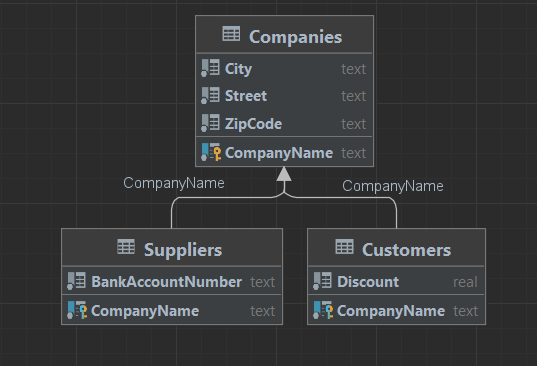
Aby zastosować tę strategię, wprowadzamy anotację [Table(„*nazwa\_tabeli*”] przed nazwą klas dziedziczących. Reszta kodu pozostaje taka sama.

## Klasa program

Kod funkcji nie zmienił się względem poprzedniego podejścia.

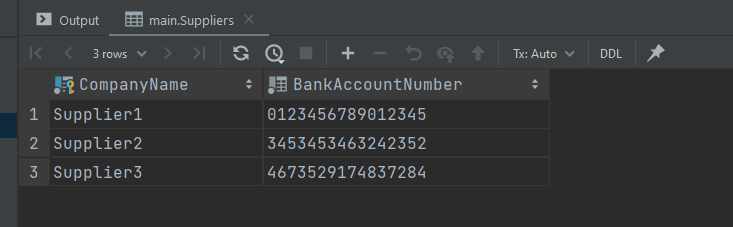
## Efekt

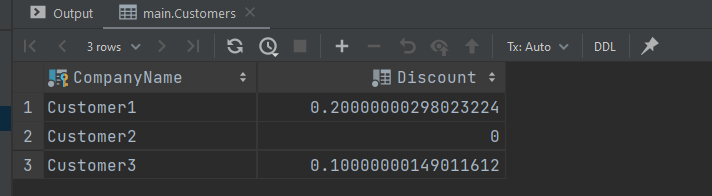
Po wykonaniu w bazie istnieją 3 tablice, osobna dla każdej klasy.

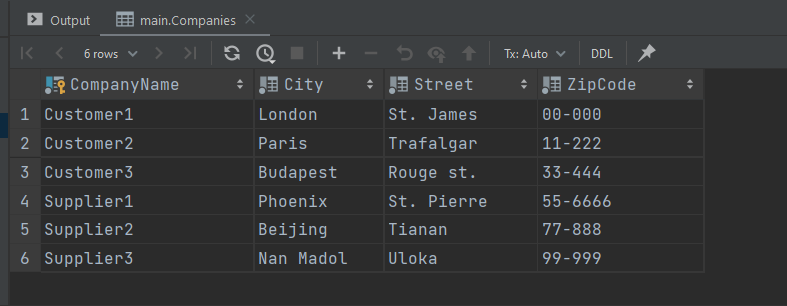


Schemat tabel w bazie danych dokładnie odzwierciedla hierarchię dziedziczenia schematu z polecenia.

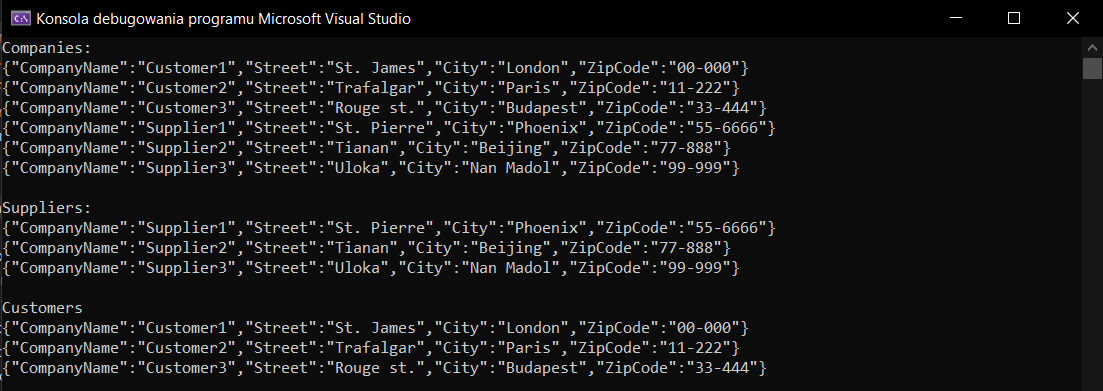
Zawartości tablic:







## Pobieranie i wypisywanie danych



# Mapowanie dziedziczenia: Table-per-concrete-type

## Nadpisujemy metodę OnModelCreation z użyciem FluentApi w kontekście ProductContext w następujący sposób:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Company>().UseTpcMappingStrategy();

}

## Kontekst

Nadpisujemy metodę OnModelCreating i używamy odpowiedniej metody dla encji Company

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Company>().UseTpcMappingStrategy();

}

## Klasy

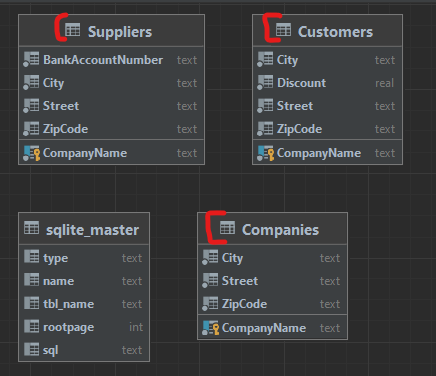
### Usuwamy anotacje [Table(„*nazwa\_tabeli*”)] z klas Customer i Supplier

## Klasa program

Bez z mian w porównaniu z poprzednim

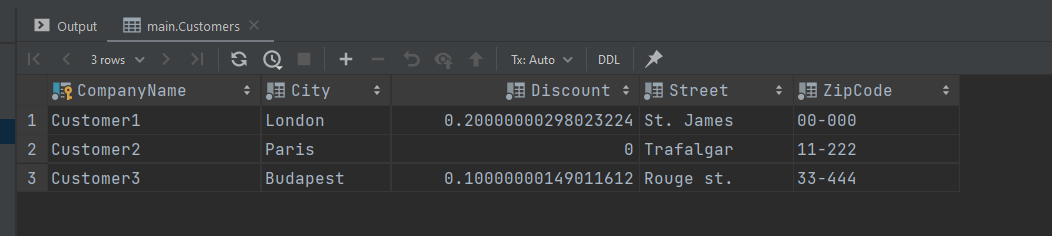
## Efekt

Schemat po wykonaniu:



Podejście Table-Per-Concrete-Type powoduje, że tablice Suppliers i Customers mają wszystkie kolumny, które posiada tablica Companies. Istnieje też osobna tablica Companies, zawierająca obiekty, które nie są klasy Customer ani Supplier (polecenie ani diagram UML nie sugerowały, że klasa ta powinna być abstrakcyjna, więc jest możliwość stworzenia jej instancji). Gdyby uczynić klasę Company abstrakcyjną, wtedy podejście Table-Per-Concrete-Type nie wygenerowałoby tablicy Companies.

Zawartości tablic:



Obraz zawierający tekst

Opis wygenerowany automatycznie

Obraz zawierający tekst

Opis wygenerowany automatycznie

## Funkcja do pobrania i wypisania danych

static void QuerryFromDatabase()

{

ProductContext productContext = new ProductContext();

var supp\_query = from supp in productContext.Suppliers

select supp;

foreach (var supplier in supp\_query)

{

Console.WriteLine(supplier);

}

var cust\_query = from cust in productContext.Customers

select cust;

foreach (var cust in cust\_query)

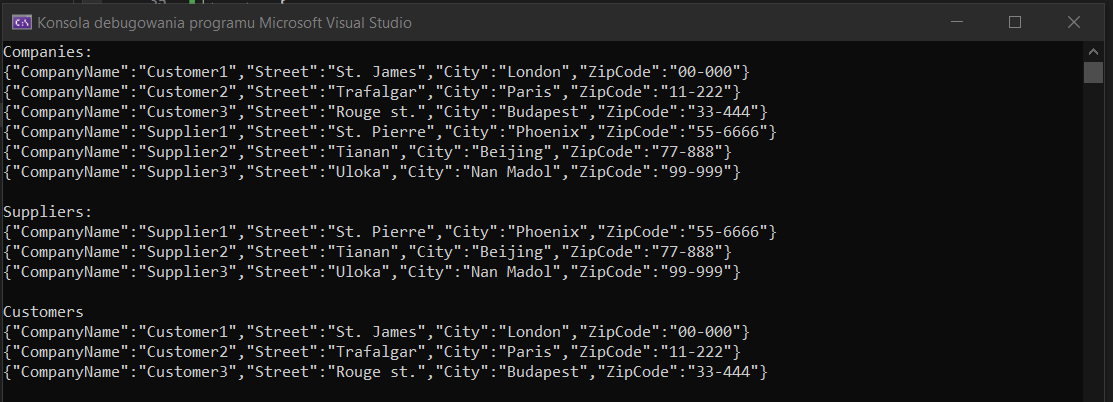
{

Console.WriteLine(cust);

}

}

## Pobranie i wypisanie danych



# Porównanie mapowań

## table-per-hierarchy

Jest to domyślna strategia w Entity Frameworku. Polega ona na, jak widać w punkcie 5., na umieszczeniu wszystkich obiektów w jednej tablicy. Typy obiektów możemy rozróżniać po kolumnie Discriminator (nazwę tę można zmienić w razie potrzeby). Zaletą tego podejścia jest stosunkowo łatwy konceptualnie model powstałej bazy. Wadą jest to, że powstała baza jest nieznormalizowana, a do tego tablica zawierająca obiekty z hierarchii może zawierać bardzo wiele wartości null.

## Table-per-Type

W tym podejściu tworzona jest osobna tabela dla każdej z klas (także abstrakcyjnych). Tabela każdej klasy zawiera tylko kolumny (pola) zadeklarowane przez siebie, a tabele klas dziedziczących odnoszą się do swojej „dziedziczonej części” przez klucz obcy do odpowiedniej encji w tabeli „nadrzędnej”.

Zaletą tego podejścia jest to, że powstała w ten sposób baza danych będzie znormalizowana i nie będą występować wartości null w kolumnach, których poszczególne encje nie potrzebowały (bo tutaj takie kolumny nie istnieją). Dodatkowo zmodyfikowanie takiej bazy sprowadzi się dodania kolejnej tablicy z kluczami obcymi to tabeli nadrzędnej.

## Table-per-concrete-type

Podejście to jest podone do table-per-type, lecz nie są tworzone tabele dla klas abstrakcyjnych. Jeśli zatem mamy klasy dziedziczące po klasie abstrakcyjnej, to własności tej klasy będą „wbudowane” niezależnie w każdą klasę dziedziczącą (o ile one same nie są abstrakcyjne). Pomiędzy takimi klasami o wspólnym abstrakcyjnym przodku nie ma żadnej relacji (mają jedynie podobne kolumny).