1. *ADO* .*NET* (*ActiveX Data Objects* .*NET*) является набором классов, реализующих программные интерфейсы для облегчения подключения к базам данных. Не зависит от бд, это только интерфейс. Классы подсоединенных объектов обеспечивают установление соединения с базой данных и управление базой со стороны приложения. Раньше всегда использовался подсоеденный режим, но это стало не эффективно. Сейчас соединение устанавливается лишь на то время, которое необходимо для выполнения операции. Доступ к данным в ADO.NET основан на использовании двух компонентов:

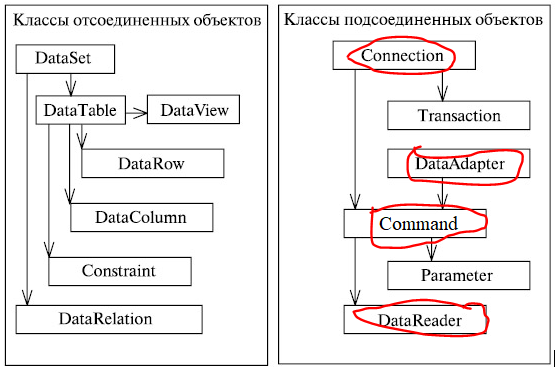
* провайдера данных (Data Provider), выполняющего функции посредника при

взаимодействии программы и баз данных;

* набора данных — объекта DataSet, в котором данные хранятся на локальном

компьютере.

*2.* классы отсоединенных объектов обеспечивают сохранение, использование и преобразование полученной от базы данных информации на стороне приложения.

Хранимая процедура может ничего не возвращать.

3. **DataSet** получает данные из базы данных через SqlDataAdapter. С DataSet мы можем локально работать с этими данными вне зависимости от наличия подключения. Не обязательно вообще иметь БД, мы можем использывать DataSet просто для удобства работы с данными. DataSet содержит таблицы, представленные объектами DataTable (DataColumn, DataRow). Clear(), Copy()

 DataSet bookStore = new DataSet("BookStore");

 DataTable booksTable = new DataTable("Books");

 bookStore.Tables.Add(booksTable);

 DataColumn idColumn = new DataColumn("Id", Type.GetType("System.Int32"));

booksTable.Columns.Add(idColumn);

 DataRow row = booksTable.NewRow();

 row.ItemArray = new object[] { null, "Война и мир", 200 };

 booksTable.Rows.Add(row);

4. **DataReader (DbDataReader)** — четает по одной записи – курсор. Когда создается, курсор ставится перед первой строкой. Close(). GetValue()

SqlConnection con = new SqlConnection(connectionString);

SqlCommand command = new SqlCommand();

command.Connection = con;

command.CommandType = CommandType.Text;

command.CommandText = "SELECT \* FROM airport";

string result = "";

con.Open();

SqlDataReader reader = command.ExecuteReader();

while (reader.Read())

{

result = reader.GetValue(0) + " " + reader.GetValue(1);

listBox1.Items.Insert(0, result);

}

reader.Close();

**DataAdapter (DbDataAdapter)** — заполняет отсоединенный объект DataSet или DataTable и обновляет его содержимое. DataAdapter предоставляет мост между объектом DataSet и источником данных. DataAdapter использует объекты Command (**update**) для выполнения команд SQL на источнике данных, для загрузки DataSet с данными и согласования изменений данных, выполненных в DataSet, вновь с источником данных.

 string connectionString = "";

            string sql = "SELECT \* FROM Users";

            using (SqlConnection connection = new SqlConnection(connectionString))

            {

                // Создаем объект DataAdapter

                SqlDataAdapter adapter = new SqlDataAdapter(sql, connection);

                // Создаем объект DataSet

                DataSet ds = new DataSet();

                // Заполняем Dataset

                adapter.Fill(ds);

            }

5. Connection(DbConnection)

SqlConnection Con = new SqlConnection(connectionString);

Con.Open();

Con.Close();

6. SqlCommand command = new SqlCommand();

command.Connection = con;

command.CommandType = CommandType.Text; //StoredProcedure

command.CommandText = "SELECT \* FROM airport";

string result = "";

con.Open();

SqlDataReader reader = command.ExecuteReader();

* **ExecuteNonQuery**: просто выполняет sql-выражение и возвращает количество измененных записей. Подходит для sql-выражений INSERT, UPDATE, DELETE.
* **ExecuteReader**: выполняет sql-выражение и возвращает строки из таблицы. Подходит для sql-выражения SELECT.
* **ExecuteScalar**: выполняет sql-выражение и возвращает одно скалярное значение, например, число. Подходит для sql-выражения SELECT в паре с одной из встроенных функций SQL, как например, Min, Max, Sum, Count.

7. (4)

8. (4)

9. (3)

10. (3)

11. DataTable companiesTable = new DataTable("Companies");

DataTable phonesTable = new DataTable("Phones");

ForeignKeyConstraint foreignKey = new ForeignKeyConstraint(companiesTable.Columns["Id"], phonesTable.Columns["CompanyId"])

{

    ConstraintName = "PhonesCompaniesForeignKey",

    DeleteRule = Rule.SetNull,

    UpdateRule = Rule.Cascade

};

ds.Tables["Phones"].Constraints.Add(foreignKey);

ds.EnforceConstraints = true;

ds.Relations.Add("PhonesCompanies", companiesTable.Columns["Id"], phonesTable.Columns["CompanyId"]);

12. DataView позволяет создавать различные представления данных, которые хранятся в DataTable.

DataView view1 = new DataView(dataset.Tables["Products"]);

view1.RowFilter = "ProductName = 'Chocolade'";

view1.Sort = **"LastName ASC, FirstName DESC";**

GridView2.DataSource = view1;

13. –

14.

* DataReader. Вытаскиваем данные из ResultSet по одному и сохраняем в DTO. Это в цикле.
* SqlDataAdapter adapter = new SqlDataAdapter(sql, con);

DataSet dataset = new DataSet();

adapter.Fill(dataset, "TableName");

15. Project->Add new DataSource->DataBase->DataSet. View->OtherWindows->DataSources. Перетаскиваем на форму. Среда сгенерирует код для DataGridView с привязкой к объекту.

16. (15)

17. ReportViewer позволяет создать выражение, которое имеет параметры, которые мы можем передать. Второй способ создания отчета – это отображение таблицы БД целиком. Copy to output derictory – always.

18 - 23. **LINQ** (Language-Integrated Query). В качестве источника данных может выступать объект, реализующий интерфейс IEnumerable (коллекции, массивы, DataSet).

DataContext db = new DataContext(Config.connectionString);

query = from j in db.GetTable<journal>()

where j.isArriving == isArriving &&

j.time > startDate

&& j.time < endDate

orderby j.time descending

join s in db.GetTable<status>() on j.statusId equals s.id

join f in db.GetTable<flight>() on j.flightId equals f.id

join a in db.GetTable<airport>() on f.airportId equals a.id

select new Recordd

(j.id, f.flight1, a.name, j.time, s.name, isArriving == 1);

foreach (Recordd journal in query)

{

listBox2.Items.Add(journal);

}

Метыды расширения

var selectedTeams = teams.Where(t=>t.ToUpper().StartsWith("Б")).OrderBy(t => t);

Добавление

DataContext db = new DataContext(connectionString);

User user1 = new User { FirstName = "Ronald", Age = 34 };

db.GetTable<User>().InsertOnSubmit(user1);

db.SubmitChanges();

Обновление

User user1 = db.GetTable<User>().FirstOrDefault();

user1.Age = 28;

db.SubmitChanges();

Удаление

var user = db.GetTable<User>().OrderByDescending(u=>u.Id).FirstOrDefault();

db.GetTable<User>().DeleteOnSubmit(user);

db.SubmitChanges();

24. Entity Framework(ORM) представляет собой более высокий уровень абстракции, который позволяет абстрагироваться от самой базы данных и работать с данными независимо от типа хранилища. LINQ для выборки данных из БД.

* **Database first**: Entity Framework создает набор классов, которые отражают модель конкретной базы данных
* **Model first**: сначала разработчик создает модель базы данных, по которой затем Entity Framework создает реальную базу данных на сервере.
* **Code first**: разработчик создает класс модели данных, которые будут храниться в бд, а затем Entity Framework по этой модели генерирует базу данных и ее таблицы

25. Entity Data Model состоит из трех уровней: концептуального, уровень хранилища и уровень маппинга.

Модель EF хранит сведения о сопоставлении классов и свойств приложений с таблицами и столбцами баз данных. Модель EF можно создать двумя способами.

* **С помощью Code First**: разработчик пишет код, чтобы указать модель. Во время выполнения EF создает модели и сопоставления на основе классов сущностей и дополнительной конфигурации модели, предоставленных разработчиком.
* **Entity Framework Designer**: разработчик рисует поля и строки для указания модели с помощью Entity Framework Designer. Результирующая модель хранится в виде XML-файла с расширением EDMX. Объекты предметной области приложения обычно создаются автоматически на основе концептуальной модели.

**26**.

: DbContext. Если у нас есть таблицы Company & User, то Context будет содержать два поля DbSet<Company> DbSet<User>

27. Context

SaveChangesAsync()

SaveChanges()

OnModelCreating(DbModelBuilder modelBuilder)

28. DbSet

Add

Create

Attach(Entity)

Find(int)

Remove

AsNoTracking<Entity>

Include – linq extension

SqlQuery

29.

Сущность содержит два типа свойств: скалярные и навигации. Атрибут навигации означает, что атрибут связан с другим объектом.

public class User

{

    public int Id { get; set; }

    public string Name { get; set; }

    public int CompanyId { get; set; }      // внешний ключ

    public Company Company { get; set; }    // навигационное свойство

}

30. Дочерний, подчиненный. Uni, Bi. One-one, one-many …

public class User

{

    public int Id { get; set; }

    public string Login { get; set; }

    public string Password { get; set; }

    public UserProfile Profile { get; set; }

}

public class UserProfile

{

    [Key]

    [ForeignKey("User")]

    public int Id { get; set; }

    public string Name { get; set; }

    public int Age { get; set; }

    public User User { get; set; }

}

31. Создавать запросы мы можем двумя способами: через операторы LINQ и через методы расщирения.

var users = (from user in db.Users.Include(p => p.Company)

                             where user.CompanyId == 1

                             select user).ToList();

var users = db.Users.Include(p=>p.Company).Where(p=> p.CompanyId == 1);

32. (31)

33.

Через навигационные свойства мы можем загружать связанные данные. И здесь у нас три стратегии загрузки:

**Eager loading** (жадная загрузка) var users = db.Users.Include(u=>u.Company).ToList();

**Explicit loading** (явная загрузка) db.Users.Where(p=>p.CompanyId==company.Id).Load();

**Lazy loading** (ленивая загрузка) **UseLazyLoadingProxies()**

34.

* Создать сущность
* Создать контекст

35. Таблицы должны называться по имени моделей во множественном числе.

По умолчанию Entity Framework уже применяет ряд конвенций, в частности, конвенцию IdKeyDiscoveryConvention, которая требует наличия свойства под названием Id или [Название\_модели]Id. Но, допустим, мы не хотим называть идентификатор id, а как-то по другому, например, "Key". В этом случае мы можем переопределить существующую конвенцию.

public class PhoneContext : DbContext

{

    public PhoneContext() : base("DefaultConnection")

    { }

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

    public DbSet<Phone> Phones { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)

    {

        modelBuilder.Properties<string>().Configure(s => s.HasMaxLength(150));

    }

}

36. Если нам необходимо, чтобы при первом обращении база данных уже была заполнена некоторыми начальными значениями, то мы можем произвести ее инициализацию.

* **CreateDatabaseIfNotExists**
* **DropCreateDatabaseIfModelChanges**
* **DropCreateDatabaseAlways**

class MyContextInitializer : DropCreateDatabaseAlways<MobileContext>

{

    protected override void Seed(MobileContext db)

    {

        Phone p1 = new Phone {Name = "Samsung Galaxy S5", Price = 14000 };

        Phone p2 = new Phone {Name = "Nokia Lumia 630", Price = 8000 };

        db.Phones.Add(p1);

        db.Phones.Add(p2);

        db.SaveChanges();

    }

}

class MobileContext : DbContext

{

    static MobileContext()

    {

        Database.SetInitializer<MobileContext>(new MyContextInitializer());

    }

    public MobileContext() : base("DefaultConnection")

    { }

    public DbSet<Phone> Phones { get; set; }

}

37. [Key], [Required], [MaxLength(20)], [NotMapped], [Table("Mobiles")], [Column("ModelName")]

38-39.

class FluentContext : DbContext

{

    public FluentContext() :base("DefaultConnection")

    {}

    public DbSet<Phone> Phones { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)

    {

        // использование Fluent API

        modelBuilder.Entity<Phone>().ToTable("Mobiles");

modelBuilder.Entity<Phone>().Property(p => p.Name).HasMaxLength(50);

modelBuilder.Entity<Phone>()

     .HasMany(p => p.Companies)

     .WithMany(c => c.Phones);

    base.OnModelCreating(modelBuilder);

    }

}

40. Миграции позволяют вносить изменения в БД при измении сущностей. Мы создали сущность, для нее сгенерилась таблица. Потом мы добавляем поле в класс. Во время запуска, EF проверяет соответсвие сущности и таблицы, если не совпадает, то накатывает миграцию. Миграция со старой схемы к новой.