/// <summary>

/// Class DebugFeatures.

/// </summary>

internal class DebugFeatures

{

/// <summary>

/// Adds the vehicles in random spaces.

/// </summary>

/// <param name="parking">The parking.</param>

/// <param name="numberOfVehicles">The number of vehicles.</param>

public static void AddVehiclesInRandomSpaces(ref Parking parking, int numberOfVehicles)

{

*Random* rnd = new *Random*();

for (int i = 0; i < numberOfVehicles; i++)

{

const string chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";

string reg;

do

{

reg = new string(*Enumerable*.*Repeat*(chars, rnd.*Next*(2, 8))

.*Select*(s => s[rnd.*Next*(s.*Length*)]).*ToArray*());

} while (parking.Find(reg) != -1);

while (parking.Add(reg, rnd.*Next*(1, 5), new string(*Enumerable*.*Repeat*(chars, rnd.*Next*(2, 8))

.*Select*(s => s[rnd.*Next*(s.*Length*)]).*ToArray*()), rnd.*Next*(0, 100)) >= 0) ;

}

}

/// <summary>

/// Adds the vehicles in order.

/// </summary>

/// <param name="parking">The parking.</param>

/// <param name="numberOfVehicles">The number of vehicles.</param>

public static void AddVehiclesInOrder(ref Parking parking, int numberOfVehicles)

{

*Random* rnd = new *Random*();

for (int i = 0; i < numberOfVehicles; i++)

{

const string chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";

string reg;

do

{

reg = new string(*Enumerable*.*Repeat*(chars, rnd.*Next*(2, 8))

.*Select*(s => s[rnd.*Next*(s.*Length*)]).*ToArray*());

} while (parking.Find(reg) != -1);

while (parking.Add(reg, rnd.*Next*(1, 5), new string(*Enumerable*.*Repeat*(chars, rnd.*Next*(2, 8))

.*Select*(s => s[rnd.*Next*(s.*Length*)]).*ToArray*())) >= 0) ;

/// <summary>

/// Class Features.

/// </summary>

internal class Features

{

/// <summary>

/// Inputs the registration.

/// </summary>

/// <returns>String.</returns>

public static *String* InputRegistration()

{

*Regex* rg = new *Regex*(@"^[a-zA-Z0-9]+$"); // adds regex for accepting a-z,A-Z,0-9

*Console*.*WriteLine*("Please input the registration number");

string reg = *Console*.*ReadLine*();

while (!rg.*IsMatch*(reg))

{

*Console*.*WriteLine*("Not a valid registration. Only A-Z and 0-9 accepted.");

*Console*.*WriteLine*("Please input the registration number");

reg = *Console*.*ReadLine*();

}

reg = reg.*ToUpper*(); //make it uppercase

return reg;

}

/// <summary>

/// Handles waiting for user to continue.

/// </summary>

public static void PressToContine()

{

*Console*.*WriteLine*("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

*Console*.*WriteLine*("Press any key to continue");

*Console*.*ReadKey*();

}

/// <summary>

/// Prints the visualisation map.

/// </summary>

/// <param name="park">The park.</param>

public static void PrintVisualisationMap(ref Parking park)

{

int[] visualMapArray = park.Visualisation();

for (int i = 0; i < visualMapArray.*Length*; i++)

{

*Console*.*BackgroundColor* = *ConsoleColor*.*DarkGray*;

if (visualMapArray[i] == 0)

{

*Console*.*ForegroundColor* = *ConsoleColor*.*Green*;

*Console*.*Write*("\u25a0 ");

}

else

{

if (visualMapArray[i] == 1)

{

*Console*.*ForegroundColor* = *ConsoleColor*.*Yellow*;

*Console*.*Write*("\u25a0 ");

}

else

{

*Console*.*ForegroundColor* = *ConsoleColor*.*Red*;

*Console*.*Write*("\u25a0 ");

}

}

///Only show 10 eachline

if ((i + 1) % 10 == 0)

{

*Console*.*WriteLine*();

}

}

*Console*.*ResetColor*();

}

/// <summary>

/// Formats the string of duration parked.

/// </summary>

/// <param name="duration">The duration.</param>

/// <returns>System.String.</returns>

public static string FormatStringOfDurationParked(*TimeSpan* duration)

{

string returnString = "";

//Print days

if (duration.*Days* == 1)

{

returnString += "1 day, ";

}

else if (duration.*Days* > 1)

{

returnString += duration.*Days* + " days, ";

}

//Print hours

if (duration.*Hours* == 1)

{

returnString += "1 hour, ";

}

else if (duration.*Hours* > 1)

{

returnString += duration.*Hours* + " hours, ";

}

if (duration.*Hours* >= 1) //print "and" if 1 hour or longer

{

returnString += "and ";

}

//Print minutes

if (duration.*Minutes* == 1)

{

returnString += "1 minute";

}

else if (duration.*Minutes* > 1 || duration.*Minutes* == 0)

{

returnString += duration.*Minutes* + " minutes";

}

//Print seconds

if (duration.*Hours* < 1) //only print seconds if under 1 hour

{

returnString += " and";

if (duration.*Seconds* == 1)

{

returnString += " 1 second";

}

else if (duration.*Seconds* > 1 || duration.*Seconds* == 0)

{

returnString += " " + duration.*Seconds* + " seconds";

}

}

return returnString;

}

}

}

using *System*;

namespace PragueParking2

{

[*Serializable*]

internal class Parking

{

/// <summary>

/// The parking spaces

/// </summary>

private ParkingSpace[] parkingSpaces;

//private int emptySpaces;

//private int partlyEmptySpaces;

/// <summary>

/// Gets the empty spaces.

/// </summary>

/// <value>The empty spaces.</value>

public int EmptySpaces

{

get

{

int emptySpaces = 0;

foreach (ParkingSpace parkingSpace in parkingSpaces)

{

if (parkingSpace.FreeSpace == 4)

{

emptySpaces += 1;

}

}

return emptySpaces;

}

}

/// <summary>

/// Gets the partly empty spaces.

/// </summary>

/// <value>The partly empty spaces.</value>

public int PartlyEmptySpaces

{

get

{

int partlyEmptySpaces = 0;

foreach (ParkingSpace parkingSpace in parkingSpaces)

{

if (parkingSpace.FreeSpace > 0 && parkingSpace.FreeSpace < parkingSpace.Size)

{

partlyEmptySpaces += 1;

}

}

return partlyEmptySpaces;

}

}

/// <summary>

/// Initializes a new instance of the <see cref="Parking"/> class.

/// </summary>

public Parking()

{

parkingSpaces = new ParkingSpace[100];

for (int i = 0; i < parkingSpaces.*Length*; i++)

{

parkingSpaces[i] = new ParkingSpace();

}

}

/// <summary>

/// Adds vehicle to the first space with room enough.

/// </summary>

/// <param name="reg">The registration number</param>

/// <param name="type">The type of vehicle</param>

/// <param name="identifier">The identifier of the vehicle.</param>

/// <returns>the index of parking space found,

/// or -2 if vehicle already in parking lot,

/// or -1 if space lacks the space</returns>

public int Add(string reg, int type, string identifier)

{

reg = reg.*ToUpper*();

int spaceFoundIndex = -1; //meaning no empty space found

if (Find(reg) != -1)

{

return -2; //meaning vehicle already in parking

}

for (int i = 0; i < parkingSpaces.*Length*; i++)

{

if (parkingSpaces[i].Add(reg, type, identifier))

{

//successfully added car

spaceFoundIndex = i;

break;

}

}

return spaceFoundIndex;

}

/// <summary>

/// Adds vehicle to a specific parking space.

/// </summary>

/// <param name="reg">The registration number</param>

/// <param name="type">The type of vehicle</param>

/// <param name="identifier">The identifier of the vehicle.</param>

/// <param name="specificSpace">The specific space to add vehicle.</param>

/// <returns>

/// the index of parking space,

/// or -1 if vehicle already in parking lot,

/// or -2 if space lacks the space

/// </returns>

public int Add(string reg, int type, string identifier, int specificSpace)

{

reg = reg.*ToUpper*();

int spaceFoundIndex = -1; //meaning no empty space found

if (Find(reg) != -1)

{

return -2; //meaning vehicle already in parking

}

if (parkingSpaces[specificSpace].Add(reg, type, identifier))

{

spaceFoundIndex = specificSpace;

}

return spaceFoundIndex;

}

/// <summary>

/// Removes vehicle with registration reg.

/// </summary>

/// <param name="reg">registration of the vehicle to remove</param>

/// <param name="durationParked">The time parked.</param>

/// <returns>

/// the space where vehicle was parked,

/// or -1 if vehicle is not in parking

/// </returns>

public int Remove(string reg, out *TimeSpan* durationParked)

{

durationParked = *TimeSpan*.*Zero*;

reg = reg.*ToUpper*();

int vehicleParkedSpaceIndex = -1; // vehicle not found

for (int i = 0; i < parkingSpaces.*Length*; i++)

{

if (parkingSpaces[i].Remove(reg, out durationParked))

{

//successfully removed vehicle

vehicleParkedSpaceIndex = i;

break;

}

}

return vehicleParkedSpaceIndex;

}

/// <summary>

/// Finds the specified reg.

/// </summary>

/// <param name="reg">The reg.</param>

/// <returns>System.Int32.</returns>

public int Find(string reg)

{

reg = reg.*ToUpper*();

for (int i = 0; i < parkingSpaces.*Length*; i++)

{

if (parkingSpaces[i] != null && parkingSpaces[i].Find(reg))

{

return i; // return index of this car in array

}

}

return -1; // not found

}

/// <summary>

/// Finds the specified reg.

/// </summary>

/// <param name="reg">The reg.</param>

/// <param name="size">The size.</param>

/// <param name="identifier">The identifier.</param>

/// <returns>System.Int32.</returns>

public int Find(string reg, out int size, out string identifier)

{

reg = reg.*ToUpper*();

for (int i = 0; i < parkingSpaces.*Length*; i++)

{

if (parkingSpaces[i] != null && parkingSpaces[i].Find(reg, out size, out identifier))

{

return i; // return index of this car in array

}

}

size = 0;

identifier = "";

return -1; // not found

}

/// <summary>

/// Moves the specified reg.

/// </summary>

/// <param name="reg">The reg.</param>

/// <param name="newSpace">The new space.</param>

/// <returns>System.Int32.</returns>

public int Move(string reg, int newSpace)

{

reg = reg.*ToUpper*();

int oldSpace = Find(reg, out int size, out string identifier);

if (oldSpace >= 0) // if exists

{

if (size <= parkingSpaces[newSpace].FreeSpace) // if parking space has room for vehicle

{

Remove(reg, out *TimeSpan* diff);

Add(reg, size, identifier, newSpace);

return 1; // means it moved vehicle

}

else

return 0; // means new space doesn't have room for vehicle

}

return -1; // means vehicle not in parking

}

public void Optimise()

{

/\*

for (int i = 0; i < parkingSpaces.Length; i++) // find trike without bike

{

if (parkingSpaces[i].FindVehicleSizes().Contains(3) && parkingSpaces[i].FreeSpace == 1)

{

}

}

\*/

}

//public string[] FindRegNumInSpace(int packingSpaceNumber)

//{

// List<Vehicle> vehiclesInSpace = new List<Vehicle>();

// ParkingSpace \_PackingSpace = new ParkingSpace();

// string[] RegNumArray = new string[4];

// for (int i = 0; i < parkingSpaces.Length; i++)

// {

// if (i == packingSpaceNumber)

// {

// RegNumArray = \_PackingSpace.FindRegNum();

// }

// }

// return RegNumArray;

//}

/// <summary>

/// Contents this instance.

/// </summary>

/// <returns>System.String.</returns>

public string Content()

{

string output = "";

for (int i = 0; i < parkingSpaces.*Length*; i++)

{

output += $"\nSpace {i + 1}: {parkingSpaces[i].Content(i)}";

}

return output;

}

/// <summary>

/// Visualisations this instance.

/// </summary>

/// <returns>System.Int32[].</returns>

public int[] Visualisation()

{

int[] visualMapArray = new int[parkingSpaces.*Length*];

for (int i = 0; i < parkingSpaces.*Length*; i++)

{

if (parkingSpaces[i].FreeSpace == 4) /// if parking space is empty

{

visualMapArray[i] = 0;

}

else

{

if (parkingSpaces[i].FreeSpace >= 1 && parkingSpaces[i].FreeSpace <= 4

) /// if parking space is between empty and full

{

visualMapArray[i] = 1;

}

else /// if parking space is full

{

visualMapArray[i] = 2;

}

}

}

return visualMapArray;

}

}

}

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

namespace PragueParking2

{

/\* Parkingspace

+int size -Sätts till 4 i när den skapas, fast värde

+int freeSpace – hur mycket finns kvar av kapaciteten

-List VehicleList

---

bool Remove(string) – ta bort bil/mc om det går

bool Find(string) – Hitta fordon baserat på regnr

string Content(int spacenumber) – Returnera vilka fordon som finns på denna plats(obs denna metod skriver EJ ut till skärmen direkt)

\*/

/// <summary>

/// Class ParkingSpace.

/// </summary>

[*Serializable*]

public class ParkingSpace

{

public int Size { get; set; }

public int FreeSpace { get; set; }

private *List*<Vehicle> VehicleList;

private Vehicle VehicleInfo = new Vehicle();

/// <summary>

/// Initializes a new instance of the <see cref="ParkingSpace"/> class.

/// </summary>

public ParkingSpace()

{

Size = 4;

FreeSpace = 4;

VehicleList = new *List*<Vehicle>();

}

///ta bort bil/mc om det går

/// <summary>

/// Removes the specified reg number.

/// </summary>

/// <param name="\_regNumber">The reg number.</param>

/// <param name="timeParked">The time parked.</param>

/// <returns><c>true</c> if XXXX, <c>false</c> otherwise.</returns>

public bool Remove(string \_regNumber, out *TimeSpan* timeParked)

{

bool result = true;

timeParked = *TimeSpan*.*Zero*;

foreach (Vehicle VehicleInfo in VehicleList.*ToList*())

{

if (VehicleInfo.RegNumber == \_regNumber)

{

VehicleList.*Remove*(VehicleInfo);

FreeSpace += VehicleInfo.VehicleSize;

timeParked =

*DateTime*.*Now*.*Subtract*(VehicleInfo

.ArrivedTime); //subtract parked time from current time to get difference

//vehicle.VehicleSize -= vehicle.VehicleSize;

result = true;

}

else

{

result = false;

}

}

return result;

}

//– Hitta fordon baserat på regnr

/// <summary>

/// Finds the specified reg number.

/// </summary>

/// <param name="\_regNumber">The reg number.</param>

/// <returns><c>true</c> if XXXX, <c>false</c> otherwise.</returns>

public bool Find(string \_regNumber)

{

//bool result = false;

foreach (Vehicle \_Vehicle in VehicleList)

{

if (\_Vehicle.RegNumber == \_regNumber)

return true;

}

return false;

}

/// <summary>

/// Finds the reg number.

/// </summary>

/// <returns>System.String[].</returns>

public string[] FindRegNum()

{

string[] RegNumArray = new string[VehicleList.*Count*];

for (int i = 0; i < VehicleList.*Count*; i++)

{

RegNumArray[i] = VehicleList[i].RegNumber;

}

return RegNumArray;

}

/// <summary>

/// Finds the vehicle sizes.

/// </summary>

/// <returns>System.Int32[].</returns>

public int[] FindVehicleSizes()

{

int[] sizeArray = new int[VehicleList.*Count*];

for (int i = 0; i < VehicleList.*Count*; i++)

{

sizeArray[i] = VehicleList[i].VehicleSize;

}

return sizeArray;

}

/// <summary>

/// Finds the specified reg number.

/// </summary>

/// <param name="\_regNumber">The reg number.</param>

/// <param name="size">The size.</param>

/// <param name="identification">The identification.</param>

/// <returns><c>true</c> if XXXX, <c>false</c> otherwise.</returns>

public bool Find(string \_regNumber, out int size, out string identification)

{

//bool result = false;

size = 0;

identification = "";

foreach (Vehicle \_Vehicle in VehicleList)

{

if (\_Vehicle.RegNumber == \_regNumber)

{

size = \_Vehicle.VehicleSize;

identification = \_Vehicle.VehicleIdentification;

return true;

}

}

return false;

}

/// <summary>

/// Contents the specified i.

/// </summary>

/// <param name="i">The i.</param>

/// <returns>System.String.</returns>

public string Content(int i)

///Returnera vilka fordon som finns på denna plats(obs denna metod skriver EJ ut till skärmen direkt)

{

string output = "";

for (int j = 0; j < VehicleList.*Count*; j++)

{

if (j > 0)

{

output += "\n ";

if (i > 8)

{

output += " ";

if (i > 98)

{

output += " ";

}

}

output += "and ";

}

output +=

$"{VehicleList[j].RegNumber}, type: {VehicleList[j].VehicleType}, {VehicleList[j].VehicleIdentification}, {VehicleList[j].ArrivedTime}";

}

return output;

}

/// <summary>

/// Adds the specified reg number.

/// </summary>

/// <param name="\_regNumber">The reg number.</param>

/// <param name="type">The type.</param>

/// <param name="identifier">The identifier.</param>

/// <returns><c>true</c> if XXXX, <c>false</c> otherwise.</returns>

public bool Add(string \_regNumber, int type, string identifier)

{

bool result = false;

switch (type)

{

case 1:

if (FreeSpace >= type)

{

VehicleInfo = new Bike(\_regNumber, identifier);

VehicleList.*Add*(VehicleInfo);

result = true;

FreeSpace -= type;

}

break;

case 2:

if (FreeSpace >= type)

{

VehicleInfo = new Motorbike(\_regNumber, identifier);

VehicleList.*Add*(VehicleInfo);

result = true;

FreeSpace -= type;

}

break;

case 3:

if (FreeSpace >= type)

{

VehicleInfo = new Tricycle(\_regNumber, identifier);

VehicleList.*Add*(VehicleInfo);

result = true;

FreeSpace -= type;

}

break;

case 4:

if (FreeSpace == type)

{

VehicleInfo = new Car(\_regNumber, identifier);

VehicleList.*Add*(VehicleInfo);

result = true;

FreeSpace -= type;

}

break;

default:

break;

}

return result;

}

}

}

/// <summary>

/// Class ReadFromBin.

/// </summary>

internal class ReadFromBin

{

/// <summary>

/// Reads an object instance from a binary file.

/// </summary>

/// <typeparam name="T">The type of object to read from the XML.</typeparam>

/// <param name="filePath">The file path to read the object instance from.</param>

/// <returns>Returns a new instance of the object read from the binary file.</returns>

public static T ReadFromBinaryFile<T>(string filePath)

{

using (*Stream* stream = *File*.*Open*(filePath, *FileMode*.*Open*))

{

var binaryFormatter = new *System*.*Runtime*.*Serialization*.*Formatters*.*Binary*.*BinaryFormatter*();

return (T) binaryFormatter.*Deserialize*(stream);

}

}

}

}

/// <summary>

/// Class Vehicles./

/// </summary>

[*Serializable*]

public class Vehicle

{

public string RegNumber { get; set; }

public string VehicleType { get; set; }

public int VehicleSize { get; set; }

public *DateTime* ArrivedTime { get; set; }

public string VehicleIdentification { get; set; }

public Vehicle(string regNumber, string vehicleIdentification)

{

RegNumber = regNumber;

VehicleIdentification = vehicleIdentification;

ArrivedTime = *DateTime*.*Now*;

}

public Vehicle()

{

}

}

/// <summary>

/// Class Car./

/// </summary>

/// <seealso cref="PragueParking2.Vehicle" />

[*Serializable*]

public class Car : Vehicle

{

public Car(string regNumber, string vehicleIdentification) : base(regNumber, vehicleIdentification)

{

VehicleSize = 4;

VehicleType = "CAR";

}

}

/// <summary>

/// Class Motorbike./

/// </summary>

/// <seealso cref="PragueParking2.Vehicle" />

[*Serializable*]

public class Motorbike : Vehicle

{

public Motorbike(string regNumber, string vehicleIdentification) : base(regNumber, vehicleIdentification)

{

VehicleSize = 2;

VehicleType = "MOTORBIKE";

}

}

/// <summary>

/// Class Bike.

/// </summary>

/// <seealso cref="PragueParking2.Vehicle" />

[*Serializable*]

public class Bike : Vehicle

{

public Bike(string regNumber, string vehicleIdentification) : base(regNumber, vehicleIdentification)

{

VehicleSize = 1;

VehicleType = "BIKE";

}

}

/// <summary>

/// Class Tricycle./

/// </summary>

/// <seealso cref="PragueParking2.Vehicle" />

[*Serializable*]

public class Tricycle : Vehicle

{

public Tricycle(string regNumber, string vehicleIdentification) : base(regNumber, vehicleIdentification)

{

VehicleSize = 3;

VehicleType = "TRICYCLE";

}

}

}

using *System*;

using *System*.*IO*;

/// <summary>

/// Functions for performing common binary Serialization operations.

/// <para>All properties and variables will be serialized.</para><para>Object type (and all child types) must be decorated with the [Serializable] attribute.</para><para>To prevent a variable from being serialized, decorate it with the [NonSerialized] attribute; cannot be applied to properties.</para>

/// </summary>

[*Serializable*]

public class WriteToBin

{

/// <summary>

/// Writes to binary file.

/// </summary>

/// <typeparam name="T"></typeparam>

/// <param name="filePath">The file path.</param>

/// <param name="objectToWrite">The object to write.</param>

/// <param name="append">if set to <c>true</c> [append].</param>

public static void WriteToBinaryFile<T>(string filePath, T objectToWrite, bool append = false)

{

using (*Stream* stream = *File*.*Open*(filePath, append ? *FileMode*.*Append* : *FileMode*.*Create*))

{

var binaryFormatter = new *System*.*Runtime*.*Serialization*.*Formatters*.*Binary*.*BinaryFormatter*();

binaryFormatter.*Serialize*(stream, objectToWrite);

}

}

}

}

/// <summary>

/// Class Program.

/// </summary>

[*Serializable*]

public class Program

{

/// <summary>

/// Defines the entry point of the application.

/// </summary>

/// <param name="args">The arguments.</param>

private static void Main(string[] args)

{

Menu start = new Menu();

}

}

}