Java for C# developers handbook

for beginners and intermediates

Author > Boban Mikšin

Version > 1.0

Table of Contents

Introduction

Java for C# developers document describes the differences between the two languages and how to easily make a transition from C# to Java without having any prior knowledge about Java language or Java platform. Versions that are considered here are .NET Framework 4.5 with C# 5.0 and Java 8.1.

For people who are familiar with Java fundamentals such as classes and types can skip those parts and check the more advanced topic Multithreading.

This document is not describing every aspect of Java language, and its primary intention is to introduce C# developers to the Java ecosystem quickly. The Android platform will not be covered.

The same chapters will exclusively target Java language or platform, so there will be no direct comparison between languages.

Framework

Both the .NET Framework and Java Platform are solving the same problems and providing a similar set of rich features. However, there are many implementation differences between them. One of the significant differences is portability, and Java is well known for its possibility to run on a different OS where .NET is limited only to the Windows (Mono and .NET Core are not considered here).

.NET Framework can be seen as a compact piece of software with rich features for building web, service, or desktop applications. Java, on the other hand, is organized in a different way where some of the features are not provided out of the box, such as *Spring* for web application development.

Here's the very brief feature comparison:

Feature	.NET	Java
OS	Windows	Multiple
Runtime	CLR	JVM
Web Server Scripting	ASP.NET WebForms	JSF
Web Framework	ASP.NET MVC	Spring (add-on)
Data Access	ADO.NET	JDBC
ORM	Entity Framework	Hibernate (add-on)

As .NET, Java also has the concept of an intermediate language which is called bytecode. Generally, for every .java file, Java compiler will create an equivalent .class file with bytecode inside.

Java Platform

People use the term Java when they want to refer to the JRE (Java Runtime Environment) and/or Java SE (Java Standard Edition). Two products implement Java Standard Edition specification, JRE and JDK (Java SE Development Kit). Depending on the needs, one of those two should be used. JRE is required to run an application, where for developing one, JDK is a must.

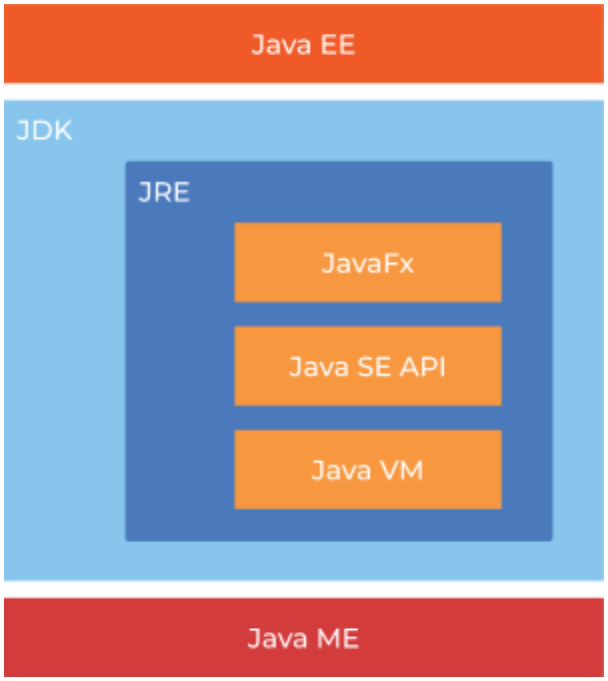


Figure 1. Java platform

The above diagram is based on the Oracle implementation of Java specifications.

- Java SDK - Java Standard Development Kit used for creating Java applications.
- JavaFX - Part of the standard JRE for building desktop and rich internet applications (RIA).
- Java EE - Java Enterprise Edition presents a superset of the standard JRE for building enterprise applications.

- Java ME - Java Micro Edition presents a subset of the standard JRE for building embedded systems.

Besides Java that is provided by Oracle, there is another Java platform called Android provided by Google. Although similar platforms, Android has some significant differences in comparison to Java. Android will not be considered in this document.

Development

IDE

In the .NET world, the most used development IDE is the Visual Studio where in Java there are three common Eclipse, NetBeans and JetBrains IntelliJ. The most similar to Visual Studio, feature equipped and preferred is the JetBrains IntelliJ.

Project

As in .NET, there are different project types in Java as well. Besides different project types, there are a couple of different ways on how the project can be organized and managed. In Visual Studio, a project can be seen as one assembly or executable where the solution is used to logically group different projects. In Java, projects can be perceived differently depending on an IDE or project type. Gradle type of project supports the concept of project and subprojects. Seeing the project from the IntelliJ perspective, a subproject will be referred to and named as a module where the project is something equal to the Visual Studio project.

Language

This section covers fundamental keywords and statements of C# programming language and describes equivalents in Java programming language. For some parts that do not exist in Java, certain alternatives will be provided.

Class

Precisely as in C#, class in Java presents one of the main programming building blocks.

- Only one class can be inherited.
- To inherit a class, an *extends* keyword should be used.
- Inherited class is usually referred to as extended or superclass.
- There is no concept of static classes in Java.
- Class is always treated as a reference type.

C#

```
public class TypeResolver : BaseResolver
{
}
```

Java

```
public class TypeResolver extends BaseResolver {
}
```

In Java, some classes are often called a *bean*. Essentially it means a class that follows the *JavaBean* standard:

- Implements the Serializable interface.
- All properties are private and exposed with appropriate getters and setters.
- Has a public constructor with no parameters.

Constructor

- Constructor, in both languages, behaves in the same way except the syntax for chaining the constructors is different.
- Chaining of constructors in Java must be done from the constructor body as opposite in C#, where chaining is done in constructor definition.
- The *super* keyword is used for calling a constructor from the extended class where *this* keyword is used for calling an overloaded constructor within the same class.

C#

```
public class BaseResolver
{
    public BaseResolver(string name)
    {
    }
}

public class TypeResolver : BaseResolver
{
    public TypeResolver()
        : this("TypeResolver", 0)
    {
    }

    public TypeResolver(string name, int type)
        : base(name)
    {
    }
}
```

Java

```
public class BaseResolver {

   public BaseResolver(String name) {
   }
}

public class TypeResolver extends BaseResolver {

   public TypeResolver() {
       this("TypeResolver", 0);
   }

   public TypeResolver(String name, int type) {
       super(name);
   }
}
```

Besides constructor, Java has another concept called instance initializer. An instance initializer is executed during the initialization of a class. One class can have multiple instance initializers, and they are executed from top to bottom. Any instance initializer is executed before a specific constructor gets called.

Java

```
public class TypeResolver {

   private final String _name;
   private final int _type;

   {
       _name = "TypeResolver";
   }

   public TypeResolver(int type) {
       _type = type;
   }
}
```

For the example above, a default constructor is not needed and as well, call to this() from each constructor.

Struct

Java has no structs. The most similar equivalence to it would be to create a class with public fields or getters and setters.

Java

```java
public class Address {

   public String Street;
   public String City;
   public String Country;
}
```

Namespace

- Java has the concept of a package that is *similar* to the namespace in C#.
- Package in Java organizes physical files. Namespace in C# organizes the program's logical parts (classes, interfaces, etc.). Although Java itself is not demanding that packages must reflect the folder structure, IDEs are the one that forces such a rule to be applied.
- Package statement is optional, but if defined, it must be the first statement in a java source file.

C#

```csharp
namespace VegaIT.CSharp.Sandbox

public class TypeResolver
{
}
```

Java

```java
package com.vegait.java.sandbox

public class TypeResolver {
}
```

Using

- What is *using* in C# it is *import* in Java.
- A package member can be imported by specifying its fully qualified name.
- Multiple package members can be imported by specifying the package name followed by an asterisk.
- Java also allows the static import of methods and constants (static fields). It can be handy in case those members are accessed frequently.

.

C#

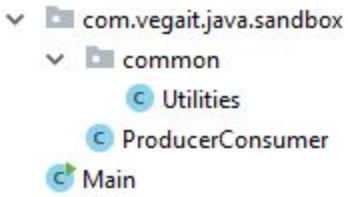
```
using System;
using RX = System.Text.RegularExpressions;

namespace VegaIT.CSharp.Sandbox
{
    public class TypeResolver
    {
        public void Register()
        {
            RX.Regex.Match("Vega IT Sourcing", "Vega");
        }
    }
}
```

Java

```
package com.vegait.java.sandbox

import java.utils.*;
import static com.vegait.java.sandbox.Utilities.isVega;

public class TypeResolver {

   public void Register(Class<?> type) {
       isPrimitiveType(type);
   }
}
```

Nesting

Java does not allow package nesting. Although most of the IDEs represent packages like they are nested, from the Java point of view, that is not the case.



Based on the example above, Java recognizes two independent packages *com.vegait.java.sandbox* and *com.vegait.java.sandbox.common*. It can be argued that there is not much of a difference in comparison to C#, but there is one big difference, and it concerns access modifiers. More on that in the next chapter.

Method

Unlike in C#, all methods in Java are virtual by default, which means they can be overridden.

C#

```
public class BaseResolver
{
    public void RegisterDefaultTypes()
    {
    }

    public virtual void Register(Guid typeId, Type type)
    {
    }
}

public class TypeResolver : BaseResolver
{
    public override void Register(Guid typeId, Type type)
    {
        base.Register(typeId, type);
    }
}
```

Java

```
public class BaseResolver {

   public void registerDefaultTypes() {
   }

   public void register(UUID typeId, Class<?> type) {
   }
}

public class TypeResolver extends BaseResolver {

   @Override
   public void registerDefaultTypes() {
       super.registerDefaultTypes();
   }

   @Override
   public void register(UUID typeId, Class<?> type) {
       super.register(typeId, type);
   }
}
```

When the method is overridden, the calling of a super method is optional and depends on the requirement. C# has a way to hide a non-virtual method from the base class by using the *new* keyword.

Parameter

One of the key points to understand in Java is how parameters are passed. In short, parameters are always passed by value. For primitive types such as int or boolean, Java will make a copy of those values and pass them to a method that is being called. Changing of values inside of the method will not change the values from the caller scope. Objects, on the other hand, are reference types, but even objects are passed by value. It means that Java will create a copy of a reference that points to that object. However, Java will not make a copy of objects or primitives inside of that object.

The same applies in C# except that C# has a way to pass a value as a reference by using the *out* keyword. Java does not have such a concept, but there is a workaround to the problem. The general rule would be to wrap a value into a class and pass an instance of that class as a parameter. Changes made on the value will be reflected from the caller.

C# has a way to pass the value type as a reference by using the *ref* or *out* keyword. More on that in the following chapters.

ref

In Java parameters as passed by value, more details on that can be read in the Parameter chapter. To change the primitive type from the calling method and have that change reflected in the caller, it is usual in Java that the primitive value gets wrapped in the class that will expose a method for updating the value.

Java

```
public class CounterWrapper {

   private int _value = 0;

   public CounterWrapper(int value) {
       _value = value;
   }

   public int getValue() { return _value; }

   public void setValue(int value) { _value = value; }
}

public class TypeResolver {

   private int _counter = 0;

   public void register(UUID typeId, Class<?> type) {
       CounterWrapper counter = new CounterWrapper(_counter);

       incrementCounter(counter);

       _counter = counter.getValue();
   }

   private static void incrementCounter(CounterWrapper counter) {
       int newValue = counter.getValue() + 1;
       counter.setValue(newValue);
   }
}
```

Java already has wrapper classes implemented for primitive types, but those classes are immutable and cannot be used in this scenario.

out

In C#, it is required that the object gets instantiated from the caller method once the parameter marked with the *out* keyword. There is nothing equivalent in Java.

Property

Java does not have properties. By having a setter and getter methods for a single field is considered as property in Java.

C#

```
public class Customer
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Java

```
public class Customer {

   private String _firstName;
   private String _lastName;

   public String getFirstName() {
       return _firstName;
   }

   public void setFirstName(String firstName) {
       _firstName = firstName;
   }

   public String getLastName() {
       return _lastName;
   }

   public void setLastName(String lastName) {
       _lastName = lastName;
   }
}
```

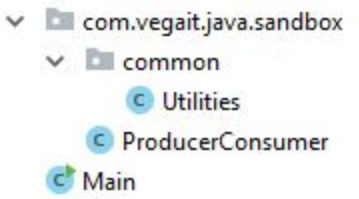
Modifiers

Access Modifiers

The following table presents a map between modifiers against types. The first column lists C# types, where the first row contains C# modifiers.

C# modifier/type	public	private	internal	protected	protected internal
class	public	N/A	no modifier [1]	N/A	N/A
interface	public	N/A	no modifier [1]	N/A	N/A
field	public	private [2]	N/A	N/A	protected [3]
method	public	private [2]	N/A	N/A	protected [3]

There is one, not that clear rule which applies to packages, namely, in the previous section namespace, it was stated that Java does not allow package nesting.



The *Utilities* package private members will not be visible from the *ProducerConsumer* class and vice versa, package private members from the *ProducerConsumer* class will not be visible from the *Utilities* class. From the C# point of view, the *com.vegait.java.sandbox* can be seen as an assembly and the *common* as a subfolder where every *internal* member from the *Utilities* class will be visible from the *ProducerConsumer* class and vice versa.

[1] accessible within the same package
[2] private members of an inner class are accessible to the outer class
[3] accessible within derived classes and anywhere in the same package

abstract

C#

```
public abstract class BaseResolver
{
    public abstract void Register(Guid typeId, Type type);
}

public class TypeResolver : BaseResolver
{
    public override void Register(Guid typeId, Type type)
    {
    }
}
```

Java

```
public abstract class BaseResolver {

   public abstract void register(UUID typeId, Class<?> type);
}

public class TypeResolver extends BaseResolver {

   @Override
   public void register(UUID typeId, Class<?> type) {
       super.register(typeId, type);
   }
}
```

const

Not the exact equivalent exists in Java, but the closest one would be the use of *static* and *final* keywords.

C#

```
public const Guid DefaultType = new Guid("484D77F3-AD41-4506-9F00-911388B49972");
```

Java

```
public static final UUID DefaultType =
UUID.fromString("484D77F3-AD41-4506-9F00-911388B49972");
```

delegate

Delegate as the method reference type in C# is also seen as a single method interface. Equivalent to a single method interface in Java would be the functional interface.

event

Java doesn't have shorthanded syntax for events like C# does. There are two ways how events can be implemented, one is using the Observable Pattern, and the second is the functional interface.

C#

```
public class TypeResolver
{
    public delegate void TypeRegisterHandler(Guid typeId, Type type);

    public event TypeRegisterHandler TypeRegistered;

    public void Register(Guid typeId, Type type)
    {
        OnTypeRegistered(typeId, type);
    }

    private void OnTypeRegistered(Guid typeId, Type type)
    {
        if (TypeRegistered != null)
            TypeRegistered(typeId, type);
    }
}

class Program
{
    static void Main()
    {
        TypeResolver typeResolver = new TypeResolver();
        typeResolver.TypeRegistered += TypeResolverOnTypeRegistered;
    }

    private static void TypeResolverOnTypeRegistered(Guid typeId, Type type)
    {
        Console.WriteLine("TypeId: {0}; Type: {1}", typeId, type);
    }
}
```

Observable pattern implementation.

Java

```
public interface TypeRegisterEvent {
   void register(UUID typeId, Class<?> type);
}

public class TypeResolver {

   private Set<TypeRegisterEvent> _listeners = new HashSet<>();

   public void addTypeRegisterEventListener(TypeRegisterEvent listener) {
       _listeners.add(listener);
   }

   public void register(UUID typeId, Class<?> type) {
       for (TypeRegisterEvent listener : _listeners) {
           listener.register(typeId, type);
       }
   }
}

public class Program implements TypeRegisterEvent {

   public void run() {
       TypeResolver typeResolver = new TypeResolver();
       typeResolver.addTypeRegisterEventListener(this);
   }

   @Override
   public void register(UUID typeId, Class<?> type) {
       System.out.println("TypeId: " + typeId + "; Type: " + type);
   }
}
```

Functional Interface implementation.

Java

```
public class TypeResolver {

   @FunctionalInterface
   public interface TypeRegisterEvent {
       void register(UUID typeId, Class<?> type);
   }

   private Set<TypeRegisterEvent> _listeners = new HashSet<>();

   public void addTypeRegisterEventListener(TypeRegisterEvent listener) {
       _listeners.add(listener);
   }

   public void register(UUID typeId, Class<?> type) {
       for (TypeRegisterEvent listener : _listeners) {
           listener.register(typeId, type);
       }
   }
}

public class Program {

   public void run() {
       TypeResolver typeResolver = new TypeResolver();
       typeResolver.addTypeRegisterEventListener(this::register);
   }

   public void register(UUID typeId, Class<?> type) {
       System.out.println("TypeId: " + typeId + "; Type: " + type);
   }
}
```

The significant difference is that the Program does not implement the TypeRegisterEvent interface but instead utilizes lambda expressions and method references. More details on the function interface can be found in the Functional interface chapter.

override

See Method chapter.

virtual

See Method chapter.

sealed

- For classes, the equivalent in Java is *final*.
- For C# sealed virtual methods, the equivalent in Java is *final*. In Java, a method is overridable by default unless marked with the *final*.

C#

```
public class Base
{
   protected virtual void Foo() { }
}

public sealed class TypeResolver : Base
{
   sealed protected override void Foo() { }
}
```

Java

```
public class Base {
   protected void Foo() {
   }
}

public final class TypeResolver extends Base {

   @Override
   protected final void Foo() {
   }
}
```

readonly

- For fields, the equivalent in Java is *final*.
- A variable in Java can also be marked as *final*.

C#

```
public class TypeResolver
{
   private readonly int _foo = 10;
   public readonly int Bar = 20;
}
```

Java

```
public class TypeResolver {

   private final int _foo = 10;
   public final int Bar = 20;
}
```

static

- Classes in Java cannot be declared as *static* unless the class is nested.
- Java doesn't support static constructors, but there is an alternative using the class initializer.

C#

```
public static class TypeResolver
{
    private static int _foo = 10;
    private static int _bar;

    static TypeResolver()
    {
        _bar = 20;
    }

    public static void Register(Guid typeId, Type type)
    {
    }
}
```

Java

```
public final class TypeResolver {

   private static int _foo = 10;
   public static int _bar;

   static {
       _bar = 20;
   }

   private TypeResolver() {
   }
}
```

volatile

More on the impact and whether it should be used can be read in the volatile chapter.

using

The concept in Java is called try-with-resources, which uses a slightly modified try statement. For resources to be automatically disposed at the end of the try block statement, it has to implement AutoCloseable or Closeable interface. More on that can be read in the IDisposable chapter.

Catch and *finally* blocks are optional.

C#

```
using (SqlConnection connection1 = new SqlConnection())
using (SqlConnection connection2 = new SqlConnection())
{
    // perform some work
}
```

Java

```
try (SqlConnection connection1 = new SqlConnection();
     SqlConnection connection2 = new SqlConnection()) {
       // perform some work
}
```

Throwing an exception from the try block is suppressing the exceptions thrown from the close method. More on exceptions and suppressed exceptions can be read in Exceptions and Exception Handling chapter.

IDisposable

Java, just like .NET, has a garbage collector (GC), which is responsible for automatic freeing up of memory from unused objects. Typically most of the objects are automatically picked by the GC, where for particular objects, it has to be explicitly defined when they should be destroyed. The notion of removing an unused object in .NET is called disposing, whereas in Java is called closing.

Two interfaces can be implemented for explicit resource closing, *Closeable*, or *AutoCloseable*. The *Closeable* throws *IOException*, where *AutoCloseable* throws *Exception*. In most cases, when there are no explicit I/O operations, the *AutoCloseable* interface is more suitable to implement.

C#

```
public class TypeResolver : IDisposable
{
    public void Dispose()
    {
    }
}
```

Java

```
public class TypeResolver implements AutoCloseable {

   @Override
   public void close() {
   }
}
```

Java does not support destructors and does not have a way to suppress the garbage collector finalize like in C#. There is a finalize method, though, that can be overridden. The finalize method gets called on an object by the JVM once it determines that there are no references to that object. When working with unmanaged resources such as explicit I/O transactions, it makes sense to override the finalize method and do the cleanup of those unmanaged resources.

Types

The concept of types is similar in both languages, but there are some minor and some significant differences when it comes to usage and implementation.

Primitive Types

Primitive types are almost the same, with few exceptions. Java does not recognize unsigned types, as C# does. In C#, we can define a number by using either *int* or *Int32,* where *Int32* is just a synonym in Java; however, *int* is treated as value type where *Integer* as its *equivalent* is a class or reference type. Those equivalent classes are also called wrapper classes, and basically, they operate by boxing a value type. Since the wrapper class is a reference type, it can be presented as a null value, so it is essential not to mix *int* with *Integer* types together when writing the code.

The following table presents the map between primitive types.

C# (type/alias)	Java (type/wrapper)
bool / System.Boolean	boolean / java.lang.Boolean
byte / System.Byte	byte / java.lang.Byte
short / System.Int16	short / java.lang.Short
int / System.Int32	int / java.lang.Integer
long / System.Int64	long / java.lang.Long
float / System.Single	float / java.lang.Float
double / System.Double	double / java.lang.Double
char / System.Char	character / java.lang.Character

Although not a primitive type per definition, the *decimal* is another type that exists in both languages. It maps to *java.math.BigDecimal* wrapper class.

Reference Types

These types in Java are based on a specific class. It means that the reference type variable holds the address of an object where the value type contains the actual value.

The following table presents the map between some reference types.

C# (type/alias)	Java (type/wrapper)
object / System.Object	N/A / java.lang.Object
string / System.String	N/A / java.lang.String
N/A / System.DateTime	N/A / java.lang.Date
N/A / System.Guid	N/A / java.lang.UUID

Enumeration Types

The enum type is implicitly inheriting *System. Enum* class or *java.lang.Enum* class in Java. Unlike in C#, Java does not have a notion of an underlying type. Instead, enum type in Java is more similar to a standard class, which gives more flexibility in the end.

C#

```
public enum VersionStatus
{
	Edit,
	ToBeApproved,
	Released,
	Archived,
	Rejected,
	Revoked,
	Deleted
}
```

Java

```
public enum VersionStatus {
   Edit(0),
   ToBeApproved(1),
   Released(2),
   Archived(3),
   Rejected(4),
   Revoked(5),
   Deleted(6);

   private final int value;

   VersionStatus(int value) {
       this.value = value;
   }

   public int getValue() {
       return this.value;
   }
}
```

Comparing Values

One of the significant differences in Java is value comparison. In C#, it is common to use == operator, even though the usage of the *Equals* method is recommended. In most cases == will behave as expected. In Java, the == operator in most cases will produce different results; thus, it is recommended to use the *equals* method whenever possible instead of the == operator. The golden rule is to use the *equals* method for every non-primitive type value comparison.

Java

```
String val1 = "Java";
String val2 = "Java";
String val3 = new String("Java");
String val4 = new String("Java");

val1 == val2 // TRUE

val3 == val4 // FALSE

val2 == val3 // FALSE

val3.equals(val4) // TRUE
```

String comparison is more difficult to grasp due to a JVM optimization, which replaces the same literals with the same reference object, thus making it appear that == operator works.

Besides the *String* type, there are specific comparison rules with the wrapper classes mentioned in the primitive types chapter.

Java

```
Integer x1 = 100;
Integer y1 = 100;

Integer x2 = 200;
Integer y2 = 200;

x1 == y1 // TRUE
x1.equals(y1) // TRUE

x2 == y2 // FALSE
x2.equals(y2) // TRUE
```

In this example, comparison outcome depends on the value range, again due to a JVM optimization.
For certain types and value ranges, the same wrapper class instance is returned.

Type	Range
byte	from -128 to 127
short	from -128 to 127
int	from -128 to 127
char	from u0000 to u00ff
boolean	true, false

Operators overloading in Java is not supported.

Nested Types

Although both languages support nested classes, Java, by default, promotes access to all outer class members from the non-static inner class.

C#

```
public class Outer
{
    private string _value = "Vega";

    private class Inner
    {
        private readonly Outer _outer;

        public Inner(Outer outer)
        {
            _outer = outer;
        }

        public void ExecuteInner()
        {
            Console.WriteLine(_outer._value);
        }
    }

    public void Execute()
    {
        var inner = new Inner(this);
        inner.ExecuteInner();
    }
}

var outer = new Outer();
var inner = new Outer.Inner(outer);
```

Java

```
public class Outer {
   private String value = "Vega";

   class Inner {
       void ExecuteInner() {
           System.out.println(value);
       }
   }

   void Execute() {
       Inner inner = new Inner();
       inner.ExecuteInner();
   }
}

Outer outer = new Outer();
Outer.Inner inner = outer.new Inner();
```

Particular types of inner classes in Java are called local classes and anonymous classes and both are described in the anonymous type chapter.

Anonymous Type

Java does not have a direct equivalent for the anonymous type like C# does. A similar result can be achieved with the local class.

C#

```
var val = new { Number = 1, Text = "Vega" };

Console.WriteLine(val.Number);
Console.WriteLine(val.Text);
```

Java

```
public static void main(String [] args) {

   class Anonymous {
       int number = 1;
       String text = "Vega";
   }

   Anonymous val = new Anonymous();

   System.out.println(val.number);
   System.out.println(val.text);
}
```

Anonymous classes in Java are like local classes but without names. They allow us to declare and instantiate a class at the same time.

Java

```
public interface TypeResolver {

   void Execute ();
}

TypeResolver typeResolver = new TypeResolver() {

   public void Execute() {
   }
};

typeResolver.Execute();
```

Anonymous class definition is an expression; thus, it must be part of a statement.

Interface

Interface, in a nutshell, is the same in both languages. However, Java has some additional features that do not exist in C# as such.

C#

```
public interface ITypeResolver
{
    void Execute();
}
```

Java

```
public interface TypeResolver {

   String Company = "VegaIT";

   void Execute ();
}
```

Apart from abstract methods, an interface in Java can have constant declarations.

More on the naming interfaces can be read in the naming convention chapter.

Functional interface

Functional interface is more like a convention where it is considered that a functional interface is the one that has only one abstract method defined. Usually, those types of interfaces are annotated with *@Functionalinterface*. They typically are not explicitly implemented but created using the lambda expressions. If the interface is not explicitly annotated with the *@Functionalinterface* annotation, but it meets the definition of a functional interface, it will be considered as such by the compiler.

Java

```
@FunctionalInterface
public interface TypeResolver {

   void Execute ();
}
```

This type of interface looks similar to the anonymous class, but the anonymous class is not bound to a single abstract method definition.

Default method

Default method is a unique feature that does not exist in C#. It allows an interface to implement a method. The main benefit of using the default method is binary compatibility, where implementations that are using an old version of the interface will still be compliant, even without recompilation.

Java

```
public interface TypeResolver {

   default void Sync () {
       // some default implementation here
   }
}
```

The default method is not required to be implemented, but it can be redeclared, which makes it an abstract method or redefine it, which overrides it.

Static method

Static method is another unique feature. It allows an interface to declare a static method that is shared with the rest of the static methods of a class that implements the interface.

Java

```
public interface TypeResolver {

   static long Recalculate (long x) {
       // some recalculation logic here
   }
}
```

Lambda expression

Lambdas are similar in both languages. In Java, a lambda can be seen as a shorthanded way of implementing the functional interface.

C#

```
public static void Print(List<Customer> customers, Func<Customer, bool> tester) {
    foreach (Customer customer in customers) {
        if (tester(customer)) {
            Console.WriteLine(customer.Name);
        }
    }
}

Print(customers, customer => customer.Name.EndsWith("Doe"));
```

Java

```
public static void print(List<Customer> customers, Predicate<Customer>
tester) {
   for (Customer customer : customers) {
       if (tester.test(customer)) {
           System.out.println(customer.getName());
       }
   }
}

print(customers, customer -> customer.getName().endsWith("Doe"));
```

In both languages, lambdas are the way to represent code as data, so in essence, they represent concrete objects. In C#, most of the lambdas correspond to *Action* and *Func* delegate types, wherein Java those types can be found in the *java.util.function* package.

Method group

Lambda expressions can be written in the shorter form if the conditions are met. That form in C# is called method group wherein Java it is called method reference.

C#

```
private static bool EndsWithDoe(Customer customer)
{
    return customer.Name.EndsWith("Doe");
}

Print(customers, EndsWithDoe);
```

Java

```
public class Customer {

   public static boolean endsWithDoe(Customer customer) {
       return customer.getName().endsWith("Doe");
   }
}

print(customers, Customer::endsWithDoe);
```

Attribute

Attribute in Java is called annotation. The annotation type is interface as opposed to attribute in C# that is of class type. In Java, it is possible to use annotation on annotation.

C#

```
[AttributeUsage(AttributeTargets.Class, Inherited = true, AllowMultiple = false)]
public class TypeIdAttribute : Attribute
{
    public TypeIdAttribute(string id)
    {
    }

    public Guid Id { get; private set; }
}
```

Java

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface EntityTypeId {

   String value();
}
```

The @ symbol denotes that the interface is of annotation type.

Generics

Generics in both languages are implemented with the same intention, specifying a generic type without knowing the actual type at the compile time. Java supports only derivation constraints.

C#

```
public abstract class Message<T>
    where T : Header
{
    protected Message(T header)
    {
        Header = header;
    }

    public T Header { get; private set; }
}
```

Java

```
public abstract class Message<T extends Header> {

   private final Class<? extends Header> _headerType;

   protected <T extends Header> Message(Class<T> headerType) {
       _headerType = headerType;
   }

   public final <T extends Header> Class<? extends Header> getHeaderType() {
       return _headerType;
   }
}
```

Implementation of generics is a bit different in Java, where generic type information is not available in the runtime. That is called *type erasure*, which doesn't allow read of such information using reflection. In cases where generic type information is required at runtime, it has to be explicitly saved, generally in some internal class field.

Covariance and contravariance

Another difference is covariance and contravariance and where they are specified. In C#, they are specified at the definition-site, wherein Java they are specified at the use-site.

Java

```
List<Integer> list1 = new ArrayList<Integer>();
List<?> list2 = new ArrayList<Integer>();
List<? extends Number> list3 = new ArrayList<Integer>();
List<? super Integer> list4 = new ArrayList<Number>();
```

LINQ

Stream in Java is the closest match to the generic query language that exists in C#. The *stream* method can be invoked on all objects that implement the *Collection* interface.

C#

```
List<string> does = customers
    .Where(c => c.Name.EndsWith("Doe"))
    .Select(c => c.Name)
    .ToList();
```

Java

```
List<String> does = customers
        .stream()
        .filter(c -> c.getName().endsWith("Doe"))
        .map(c -> c.getName())
        .collect(Collectors.toList());
```

Some of the stream methods, such as *findFirst*, which is equivalent to the *First* method in C#, return *Optional<T>*.

Reusing an already created and called *stream* results in an *IllegalStateException*.

Expression trees are not supported in Java.

Reflection

Java supports reflection at certain extent as C# does.

C#

```
PropertyInfo propertyInfo = typeof(Customer).GetProperty("Name");
propertyInfo.SetValue(customer, "Doe");
```

Java

```
Method method = Customer.class.getDeclaredMethod("setName", String.class);
method.setAccessible(true);
method.invoke(customer, "Doe");
```

Reading information about generic types and method parameters is not supported. Java implements so-called type erasure, so to have information available in the runtime, it is required that such information gets explicitly stored.

Serialization

Java supports binary serialization out of the box.

C#

```
[Serializable]
public class Entity
{
    private readonly string _name;

    [NonSerialized]
    private readonly string _secret;

    public Entity(string name, string secret)
    {
        _name = name;
        _secret = secret;
    }

    public override string ToString()
    {
        return String.Format("Name: '{0}'; Secret: '{1}'", _name, _secret);
    }
}

using (Stream stream = File.Create("entity.bin"))
{
    BinaryFormatter formatter = new BinaryFormatter();
    formatter.Serialize(stream, new Entity("Doe", "a"));
}

using (Stream stream = File.OpenRead("entity.bin"))
{
    BinaryFormatter formatter = new BinaryFormatter();
    Entity entity = (Entity) formatter.Deserialize(stream);
}
```

Java

```
public class Customer implements Serializable {
   private static final long serialVersionUID = 7526472295622776147L;

   private String name;

   transient private String secret;

   public Customer(String name, String secret) {
       this.name = name;
       this.secret = secret;
   }

   @Override
   public String toString() {
       return String.format("Name: '%s'; Secret: %s", name, secret);
   }
}

FileOutputStream output = new FileOutputStream("entity.ser");
ObjectOutputStream serializer = new ObjectOutputStream(output);
serializer.writeObject(new Customer("Doe", "a"));
serializer.flush();

FileInputStream input = new FileInputStream("entity.ser");
ObjectInputStream deserializer = new ObjectInputStream(input);
Customer customer = (Customer)deserializer.readObject();
```

Fields that are marked with the *transient* keyword are ignored during serialization. As part of the best practice, it is recommended to always include a *serialVersionUID* field for the sake of compatibility and security.

JSON serialization is not supported, but few popular libraries are used, such as *Jackson* (also supports other formats such as XML) and *Gson*.

XML serialization is also supported out of the box by using either *JAXB* (Java Architecture for XML Binding) or *XMLEncoder/XMLDecoder* classes.

Collections

C# and Java are quite similar in the set of data structures that they provide.

The closest equivalent of *IEnumerable<T>* in Java is *Iterator<E>*. However, in Java, Iterator supports item removing that is not possible in C#. Unlike in C#, LINQ expressions can be called on the *IEnumerable* type and its descendants wherein Java, the stream function, is available for the *Collection* type and its descendants.

Table with commonly used types and their equivalents:

C#	Java
List<T>	ArrayList<E>
HashSet<T>	HashSet<E>
Dictionary<TKey, TValue>	HashMap<K, V>

Naming Convention

Package name provides typically the name of the organization followed by the project-specific details, usually reversed domain name (e.g., *com.vegait.java*).

Method names follow the camelCase notation.

Interfaces are not prefixed with *I* just like in C# and instead looks like a standard class in C#. A single class that implements an interface is typically suffixed with *Impl*.

Java

```
public interface TypeResolver {
   void execute();
}

public class TypeResolverImpl implements TypeResolver {
   @Override
   public void execute() {
   }
}
```

There are two official documents on the naming convention provided by Oracle and Google.

Exceptions and Exception Handling

Java compiler recognizes two types of exceptions, checked and unchecked. Java VM treats all exceptions as unchecked exceptions. The compiler checks checked exceptions, and the developer must handle them, and there are two ways for that. The first is to define it using the *throws* keyword, and the second is to catch it without rethrowing it.

Java

```
public Object readResolve() throws ObjectStreamException {
   return valueOf(ordinal());
}


public Object readResolve() {
   try {
       return valueOf(ordinal());
   } catch (ObjectStreamException ex) {
       throw new RuntimeException(ex);
   }
}
```

In case of using the *throws* keyword, a caller of the method would have to handle it. When wrapping an exception in *RuntimeException*, a caller would not be aware of such *RuntimeException* without looking into the source code.

Unchecked exceptions are based on either *Error* or *RuntimeException* classes. Every other exception is considered to be a checked exception. Exceptions in C# are equivalent to unchecked exceptions in Java. There is no concept of checked exceptions in C#.

According to the official Java documentation, catching a checked exception and throwing it as *RuntimeException* is not recommended. There is still debate in favor of both approaches, and none of them are exclusive. By looking at the most popular libraries such as *Spring*, *Hibernate*, *AWS SDK*, there is a tendency to lean more towards converting checked exceptions into *RuntimeException* exceptions.

Any exception thrown from the *finally* block will discard any exception thrown from the *try* and *catch* block.

Multithreading

Thread

There are different ways in both languages how the program can fire up a thread. In C#, a thread can execute a delegate or anything that applies to the delegate signature. In contrast, a Java thread can execute any class that implements a *Runnable* interface or anything applicable to that interface such, as a method without input parameters.

C#

```
public void Execute()
{
}

Thread thread = new Thread(Execute);
thread.Start();
```

Java

```
public void execute() {
}

Thread thread = new Thread(this::execute);
thread.start();
```

This is also called a *runnable pattern*. Besides the *start* method, there is also a *run* method, but that method should never be called because it defuses the purpose of executing the code in a separate thread. By calling the *run* method, a target (*Runnable*) will be executed synchronously from the calling thread. In both languages, the thread does not return anything.

Thread class in Java can be inherited where in C# *Thread* class is sealed. It is strongly discouraged to inherit *Thread* class.

Foreground vs. Background

By default in C#, any thread created is treated as the foreground thread, which means that the main thread will wait for its execution to finish to shut down

completely. In Java, foreground threads are called user threads, and they behave in the same way as to foreground threads in C#. Background threads are quite the opposite from the foreground threads and will be automatically terminated upon the primary thread completion. Those background threads are called daemon threads in Java. Daemon threads have lower priority over the user threads which, is not the case in C#, where both have the same priority unless the priority is explicitly set to a different priority.

C#

```
Thread thread = new Thread(Execute);
thread.IsBackground = true;

thread.Start();
```

Java

```
Thread thread = new Thread(this::execute);
thread.setDaemon(true);

thread.start();
```

Join

Both languages have the same syntax when it comes to blocking a thread.

C#

```
Thread thread = new Thread(Execute);
thread.Start();

thread.Join();
```

Java

```
Thread thread = new Thread(this::execute);
thread.start();

thread.join();
```

By calling the *join* method, it will block the calling thread from further execution until the thread for which the *join* was called finishes its execution.

Abort

In both languages, it is considered as bad practice to stop a thread forcibly. Although API for doing so is still available, and it is still inadequate, unreliable, and unpredictable. The recommended approach is to implement a stopping mechanism inside of a thread and allowing mechanism to be called from outside of the thread.

Exception Handling

In C#, any unhandled (uncaught) exception that was thrown from a thread will cause the process to stop. In Java, an uncaught exception that was thrown from a thread will only bring down that thread where the JVM will continue to work.

There are several ways in Java how uncaught exceptions that are thrown from a thread can be handled:

by setting an uncaught exception handler on an individual thread,

Java

```
Thread thread = new Thread(Main::execute);
thread.setUncaughtExceptionHandler(Main::ThreadExceptionHandler);
```

by setting a default (global) uncaught exception handler.

Java

```
Thread.setDefaultUncaughtExceptionHandler(Main::ThreadExceptionHandler);
```

In case of an uncaught exception, the JVM will first check if the uncaught exception handler is defined on an individual thread and call it if it is, otherwise it will look for the global uncaught exception handler. In case a thread belongs to a group, the uncaught exception handler of that group will be called.

More on exceptions and how they should be handled in general can be read in Exceptions and Exception Handling chapter.

Group

The concept of the thread group as such only exists in Java, and it is mainly introduced to manage several threads of a similar type easily.

Java

```
ThreadGroup threadGroup = new ThreadGroup("A");
Thread thread = new Thread(threadGroup, Main::execute);
```

ThreadPool

The closest implementation in Java, similar to the *ThreadPool* in C#, would be the *ExecutorService*. However, the *ExecutorService* provides much more functionality and control over threads, which is more similar to the Task in C#. *ThreadPool* in C# can be seen as a subset of Java *ExecutorService*.

C#

```
public void Execute(object state)
{
}

ThreadPool.QueueUserWorkItem(Execute);
```

Java

```
public void execute() {
}

ExecutorService executorService = Executors.newSingleThreadExecutor();
executorService.execute(Main::execute);
executorService.shutdown();
```

This is also called an *executor pattern*. In this example, *ExecutorService* will create only one thread and use it to execute any runnable that is passed through the execute method. If there are no more tasks to be executed, that single thread will still stay alive until the *shutdown* method gets called. There are two more ways how *ExecutorService* can be shut down, *shutdownNow* which will stop any executing task and shutdown and *awaitTermination,* which will wait for a specified time for tasks to complete.

For threads to be created on-demand *Executors.newCachedThreadPool* factory method should be used.

Any thread that is created by the *ExecutorService* is a user thread. There are many other ways of how *ExecutorService* can be configured. The big difference is that multiple instances of differently configured *ExecutorService* can exist, which is not the case in C#, where *ThreadPool* is a *static* class and allows minimal configuration.

ThreadPool, just like the thread, does not provide return value from a thread execution. The same is with the *ExecutorService* using the *Runnable* interface. Also, an exception that is raised from the *Runnable* will not be propagated, and there is no way to tell when the task completed. However, *ExecutorService* provides additional methods for which return value can be specified. More on that can be found in the Task chapter.

Exception Handling

As with regular threads, handling of exceptions in C# must be done within the thread itself.

By using the *ExecutorService* in combination with the *Runnable,* the only way to tell whether there was an exception is to implement the *Thread.UnchaughtExceptionHandler,* just like it is described in the Thread Exception Handling chapter.

Important is that the *ExecutorService* shutdowns at the end.

Java

```
public void execute() {
}

ExecutorService executorService = Executors.newSingleThreadExecutor();
try {
   executorService.execute(Main::execute);
} finally {
   executorService.shutdown();
}
```

Task

Task presents a higher level of abstraction over the thread. It gives much more configurable and powerful abilities when it comes to parallel execution. *TPL* (Task Parallel Library) does not have an exact and single equivalent in Java. Several implementations can be used to achieve what *TPL* provides. Not every aspect of the TPL will be covered here, only some basic ones.

C#

```
public static int Execute()
{
    return 0;
}

Task<int> task = Task<int>.Factory.StartNew(Execute);
Console.WriteLine("Task Result {0}", task.Result);
```

Java

```
public static Integer execute() {
   return 0;
}

ExecutorService executorService = Executors.newSingleThreadExecutor();
Future<Integer> task = executorService.submit(Main::execute);
System.out.println("Task result " + task.get());
```

In this case, *Main::execute* is implicitly converted to *Callable<Integer>* type, which is equivalent to *Func<int>* type in C#. By calling the *get* method, it will block the current execution until the results are provided, similar to what the *join* method of *Thread* class is doing. Although the *Future* provides more control over *Thread*, still it not fully compatible with *Task* in C#.

The closest implementation of *Task* in Java is *CompletableFuture*, which provides a model for continuation and synchronization.

C#

```
List<Task> tasks = new List<Task>();

Task task1 = Task.Factory.StartNew(Execute);
task1.ContinueWith(
    t => Console.WriteLine("Task {0} exceptions {1}", t.Id,
        String.Join(";", t.Exception.Flatten().InnerExceptions.Select(e =>
e.Message))),
    TaskContinuationOptions.OnlyOnFaulted);

Task task2 = Task.Factory.StartNew(Execute);

tasks.Add(task1);
tasks.Add(task2);

Task.Factory.ContinueWhenAll(tasks.ToArray(), _ => Console.WriteLine("Tasks
completed."));
```

Java

```
List<CompletableFuture> tasks = new ArrayList<>();

CompletableFuture task1 = CompletableFuture.runAsync(Main::execute)
       .exceptionally((e) -> {
                   System.out.println("Task exception" + e.getMessage());
                   return null;
               });

CompletableFuture task2 = CompletableFuture.runAsync(Main::execute);

tasks.add(task1);
tasks.add(task2);

CompletableFuture.allOf(tasks.toArray(new CompletableFuture[0]))
       .thenRun(() -> System.out.println("Tasks completed"));
```

There are many more options and ways of how tasks can be invoked and synchronized in parallel, which can be found online and which will not be covered further in this document.

Exception Handling

Unlike the exception handling, using the *Runnable* interface *Future* and *CompletableFuture* offer a more convenient way of handling the exceptions.

C#

```
public static void Execute()
{
}

Task task = Task.Factory.StartNew(Execute);

try
{
    task.Wait();
}
catch (AggregateException ex)
{
    Console.WriteLine("Task exceptions {0}",
        String.Join(";", ex.Flatten().InnerExceptions.Select(e => e.Message)));
}
```

Java

```
public static void execute() {
}

ExecutorService executorService = Executors.newSingleThreadExecutor();
Future task = executorService.submit(Main::execute);

try {
   task.get();
} catch (Exception ex) {
   System.out.println("Task exception " + ex.getMessage());
}
```

There is also a more convenient way of handling the exception using the *CompletableFuture*.

C#

```
Task task = Task.Factory.StartNew(Execute);
task.ContinueWith(
    t => Console.WriteLine("Task {0} exceptions {1}", t.Id,
        String.Join(";", t.Exception.Flatten().InnerExceptions.Select(e =>
e.Message))),
    TaskContinuationOptions.OnlyOnFaulted);
```

Java

```
CompletableFuture task = CompletableFuture.runAsync(Main::execute)
       .exceptionally((e) -> {
                   System.out.println("Task exception" + e.getMessage());
                   return null;
               });
```

Persistence

The closest equivalent to ADO.NET in Java would be JDBC. Usually, these days are more likely to use a different framework that is built on top of ADO.NET and JDBC, such as Entity Framework/NPoco in C# or JPA/Hibernate in Java.

C#

```
using (SqlConnection connection = new SqlConnection(@"Data Source=.;Initial
Catalog=VegaIT;Integrated Security=True"))
using (SqlCommand command = new SqlCommand("SELECT Name FROM Project", connection))
{
    connection.Open();

    using (SqlDataReader reader = command.ExecuteReader())
    {
        while (reader.Read())
        {
            Console.WriteLine("Name: {0}.", reader.GetString(0));
        }
    }
}
```

Java

```
try (Connection connection =
DriverManager.getConnection("jdbc:sqlserver://localhost:1433;database=VegaIT;integr
atedSecurity=true");
    Statement statement = connection.createStatement();) {

   resultSet = statement.executeQuery("SELECT Name FROM Project");

   while (resultSet.next()) {
       System.out.println("Name: " + resultSet.getString(1));
   }
}
```

The provided example requires the *Microsoft JDBC Driver for SQL Server* package, which can be obtained from the official Microsoft site using the following link. An alternative is to obtain it from the Maven central repository. More on Maven and how to obtain artifacts can be found in the next chapter NuGet.

NuGet

NuGet can be seen as a subset of the Apache Maven, where Maven is a complete build tool that provides a lot more than package management. Maven is designed to work independently from an IDE and can support different languages and platforms. Other tools can be used for dependency management, such as Gradle and Ivy.

nuget.org is the central repository for .NET, and its equivalent in Java is mvnrepository.com.

Every NuGet package is uniquely identified by its *id* (e.g. Newtonsoft.Json) and *version* (e.g. 12.0.3), whereas in Java dependency is uniquely identified by its *groupId* (e.g. org.springframework), *artifactId* (e.g. spring-core) and *version* (e.g. 5.2.5).

Inversion of Control Container

There are several IoC containers available for Java, and the most used one is Spring. There will be on comparison with any container available for C#.

To use Spring as the container, it is necessary to reference the Spring Context (i.e., an artifact from the Maven repository). More on artifacts and how they can be obtained can be found in the NuGet chapter.

Examples will cover the auto wiring approach, where XML configuration will not be covered. Components in Spring are referred to as beans.

Java

```
package com.vegait.java.sandbox;

public interface TypeResolverService {
}

@Service
public class TypeResolverServiceImpl implements TypeResolverService {
}

@Configuration
@ComponentScan({"com.vegait"})
public class AppConfig {
}

AnnotationConfigApplicationContext applicationContext = new
AnnotationConfigApplicationContext(AppConfig.class);

TypeResolverService typeResolverService =
applicationContext.getBean(TypeResolverService.class);
```

Configuration in this example automatically scans for annotated services and register them based on the interfaces that are implemented. Any attempt to resolve a bean, based on the *TypeResolverService* interface will result in returning a new instance of the *TypeResolverServiceImpl* class.

The following example extends *TypeResolverServiceImpl* with *ParameterService* class, which instance is explicitly registered.

Java

```
public class ParametersService {
}


@Service
@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)
public class TypeResolverServiceImpl implements TypeResolverService {

   public TypeResolverServiceImpl(ParametersService parametersService) {
   }
}

ParametersService parametersService = new ParametersService();

GenericBeanDefinition bd = new GenericBeanDefinition();
bd.setBeanClass(ParametersService.class);
bd.setInstanceSupplier(() -> parametersService);

DefaultListableBeanFactory beanFactory = new DefaultListableBeanFactory();
beanFactory.registerBeanDefinition("parametersService", bd);

AnnotationConfigApplicationContext applicationContext = new
AnnotationConfigApplicationContext(beanFactory);
applicationContext.register(AppConfig.class);
applicationContext.refresh();
```

There are many other ways and options in Spring how beans can be registered and resolved, more on that can be read on the official site.

Unit Testing

Java does not have a built-in unit test framework like .NET has (MSTest). To write unit tests, an external framework has to be installed. The most popular and frequently used is JUnit.

Java

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNull;

public class TypeResolverShould {

   @Test
   public void resolve_StringType() {
       UUID typeId = UUID.fromString("C11E3901-2A1C-4091-BCBE-049C74B5C9BB");

       TypeInfo actualType = TypeResolver.resolve(typeId);

       String value = "Test";
       Class expectedType = value.getClass();

       assertEquals(expectedType, actualType.getType());
   }
}
```

Web

The closest match of *ASP.NET MVC* in Java would be *Java EE 8 MVC*. These days it is more popular to use a third part web framework such as *Spring MVC*.

Web Service

In Java, *JAX-WS* is used for building XML/SOAP web services that must support WS-* standards. Other third party libraries can be used, such as Spring, which also supports the building of REST-based services.