WIKIPEDIA

Comparison of C Sharp and Java

This article compares two programming languages: **C#** with **Java**. While the focus of this article is mainly the languages and their features, such a comparison will necessarily also consider some features of platforms and libraries. For a more detailed comparison of the platforms, please see Comparison of the Java and .NET platforms.

C# and Java are similar languages that are typed statically, strongly, and manifestly. Both are object-oriented, and designed with semi-interpretation or runtime just-in-time compilation, and both are curly brace languages, like C and C++.

Contents

Types

Data types	Java	C#
Arbitrary-size decimals	Reference type; no operators[1]	Third party library[2]
Arbitrary-size integers	Reference type; no operators	Yes[3]
Arrays	Yes[4]	Yes
Boolean type	Yes	Yes
Character	Yes[5]	Yes
Complex numbers	Third-party library[6]	Yes
Date/time	Yes; reference type[7]	Yes; value type
Enumerated types	Yes; reference type	Yes; scalar
High-precision decimal number	No; but see 'Arbitrary-size decimals' above	128-bit (28 digits) Decimal type[8]
IEEE 754 binary32 floating point number	Yes	Yes
IEEE 754 binary64 floating point number	Yes	Yes
Lifted (nullable) types	No; but wrapper types	Yes
Pointers	No;[9] only method references[10]	Yes[11]
Reference types	Yes	Yes
Signed integers	Yes; 8, 16, 32, 64 bits	Yes; 8, 16, 32, 64 bits
Strings	Immutable reference type, Unicode	Immutable reference type, Unicode
Type annotations	Yes	Yes
Single-root (unified) type system	No; but wrapper types	Yes[12]
Tuples	No; limited 3rd party available.[13]	Yes[14]
Unsigned integers	No; but some method support.[15]	Yes; 8, 16, 32, 64 bits
Value types	No; only primitive types	Yes

Unified type system

Both languages are statically typed with class-based object orientation. In Java the primitive types are special in that they are not object-oriented and they could not have been defined using the language itself. They also do not share a common ancestor with reference types. The Java reference types all derive from a common root type. C# has a unified type system in which all types (besides unsafe pointers[16]) ultimately derive from a common root type. Consequently, all types implement the methods of this root type, and extension methods defined for the `object` type apply to all types, even primitive `int` literals and delegates. Note, that unlike Java, this allows C# to support objects with encapsulation that are not reference types.

In Java, compound types are synonymous with reference types; methods cannot be defined for a type unless it is also a *class* reference type. In C# the concepts of encapsulation and methods have been decoupled from the reference requirement so that a type can support methods and encapsulation without being a reference type. Only reference types support virtual methods and specialization, however.

Both languages support many built-in types that are copied and passed by value rather than by reference. Java calls these types primitive types, while they are called *simple types* in C#. The primitive/simple types typically have native support from the underlying processor architecture.

The C# primitive/simple types implement several interfaces and consequently offer many methods directly on instances of the types, even on the literals. The C# type names are also merely *aliases* for Common Language Runtime (CLR) types. The C# `System.Int64` type is exactly the same type as the `long` type; the only difference is that the former is the canonical .NET name, while the latter is a C# alias for it.

Java does not offer methods directly on primitive types. Instead, methods that operate on primitive values are offered through companion primitive wrapper classes. A fixed set of such wrapper classes exist, each of which wraps one of the fixed set of primitive types. As an example, the Java `Long` type is a reference type that wraps the primitive `long` type. They are *not* the same type, however.

Data types

Numeric types

Signed integers

Both Java and C# support signed integers with bit widths of 8, 16, 32 and 64 bits. They use the same name/aliases for the types, except for the 8-bit integer that is called a `byte` in Java and a `sbyte` (signed byte) in C#.

Unsigned integers

C# supports unsigned in addition to the signed integer types. The unsigned types are `byte`, `ushort`, `uint` and `ulong` for 8, 16, 32 and 64 bit widths, respectively. Unsigned arithmetic operating on the types are supported as well. For example, adding two unsigned integers (`uints`) still yields a `uint` as a result; not a long or signed integer.

Java does not feature unsigned integer types. In particular, Java lacks a primitive type for an unsigned byte. Instead, Java's `byte` type is sign extended, which is a common source of bugs and confusion.[17]

Unsigned integers were left out of Java deliberately because James Gosling believed that programmers would not understand how unsigned arithmetic works.

> In programming language design, one of the standard problems is that the language grows so complex that nobody can understand it. One of the little experiments I tried was asking people about the rules for unsigned arithmetic in C. It turns out nobody understands how unsigned arithmetic in C works. There are a few obvious things that people understand, but many people don't understand it.[9][18]

High-precision decimal numbers

C# has a type and literal notation for high-precision (28 decimal digits) decimal arithmetic that is appropriate for financial and monetary calculations.[19][20][21] Contrary to the `float` and `double` data types, decimal fractional numbers such as 0.1 can be represented exactly in the decimal representation. In the float and double representations, such numbers often have non-terminating binary expansions, making those representations more prone to round-off errors.[20]

While Java lacks such a built-in type, the Java library does feature an *arbitrary precision* decimal type. This is not considered a language type and it does not support the usual arithmetic operators; rather it is a reference type that must be manipulated using the type methods. See more about arbitrary-size/precision numbers below.

Advanced numeric types

Both languages offer library-defined arbitrary-precision arithmetic types for arbitrary-size integers and decimal point calculations.

Only Java has a data type for arbitrary precision decimal point calculations. Only C# has a type for working with complex numbers.

In both languages, the number of operations that can be performed on the advanced numeric types are limited compared to the built-in IEEE 754 floating point types. For instance, none of the arbitrary-size types support square root or logarithms.

C# allows library-defined types to be integrated with existing types and operators by using custom implicit/explicit conversions and operator overloading. See example in section *Integration of library-defined types*

Characters

Both languages feature a native `char` (character) datatype as a simple type. Although the `char` type can be used with bit-wise operators, this is performed by promoting the `char` value to an integer value before the operation. Thus, the result of a bitwise operation is a numeric type, not a character, in both languages.

Built-in compound data types

Both languages treat strings as (immutable) objects of reference type. In both languages, the type contains several methods to manipulate strings, parse, format, etc. In both languages regular expressions are considered an external feature and are implemented in separate classes.

Both languages' libraries define classes for working with dates and calendars in different cultures. The Java `java.util.Date` is a mutable reference type, where the C# `System.DateTime` is a struct value type. C# additionally defines a `TimeSpan` type for working with time periods. Both languages support date and time arithmetic according to different cultures.

User-defined value type (struct)

C# allows the programmer to create user-defined value types, using the `struct` keyword. Unlike classes and like the standard primitives, such value types are passed and assigned by value rather than by reference. They can also be part of an object (either as a field or boxed), or stored in an array without the memory indirection that normally exists for class types.

Because value types have no notion of a *`null`* value and can be used in arrays without initialization, they always come with an implicit default constructor that essentially fills the struct memory space with zeroes. The programmer can only define additional constructors with one or more arguments. Value types do not have virtual method tables, and because of that (and the fixed memory footprint), they are implicitly sealed. However, value types *can* (and frequently do) implement interfaces. For example, the built-in integer types implement several interfaces.

Apart from the built-in primitive types, Java does not include the concept of value types.

Enumerations

Both languages define enumerations, but they are implemented in fundamentally different ways. As such, enumerations are one area where tools designed to automatically translate code between the two languages (such as Java to C# converters) fail.

C# has implemented enumerations in a manner similar to C, that is as wrappers around the bit-flags implemented in primitive integral types (int, byte, short, etc.). This has performance benefits and improves interaction with C/C++ compiled code, but provides fewer features and can lead to bugs if low-level value types are directly cast to an enumeration type, as is allowed in the C# language. Therefore, it is seen as syntactic sugar.[22] In contrast, Java implements enumerations as full featured collection of instances, requiring more memory and not aiding interaction with C/C++ code, but providing additional features in reflection and intrinsic behavior. The implementation in each language is described in the table below.

	Java	C#
Definition	In Java, the enumeration type is a class, and its values are objects (instances) of that class. The only valid values are the ones listed in the enumeration. The enumeration type may declare fields, allowing each individual enumerated value to reference additional data associated uniquely with that specific value. The enumeration type may also declare or override methods, or implement interfaces.[23]	Enumerations in C# are implicitly derived from the `Enum` type that again is a value type derivative. The value set of a C# enumeration is defined by the *underlying type* that can be a signed or unsigned integer type of 8, 16, 32 or 64 bits. The enumeration definition defines names for the selected integer values.[23][24] By default the first name is assigned the value 0 (zero) and the following names are assigned in increments of 1. Any value of the underlying primitive type is a valid value of the enumeration type, though an explicit cast may be needed to assign it.
Combining	Java enumeration set and map collections provide functionality to combine multiple enumeration values to a combined value. These special collections allows compiler optimization to minimize the overhead incurred by using collections as the combination mechanism.	C# supports bit-mapped enumerations where an actual value may be a combination of enumerated values bitwise or'ed together. The formatting and parsing methods implicitly defined by the type will attempt to use these values.

In both C# and Java, programmers can use enumerations in a switch statement without conversion to a string or primitive integer type. However, C# disallows fall-throughs unless the case statement does not contain any code, as they are a main cause for hard-to-find bugs.[25] Fall-throughs must be explicitly declared using goto case[26]

Delegates, method references

C# implements object-oriented method pointers in the form of delegates. A delegate is a special type that can capture a type-safe reference to a method. This reference can then be stored in a delegate-type variable or passed to a method through a delegate parameter for later invocation. C# delegates support covariance and contravariance, and can hold a reference to any signature-compatible static method, instance method, anonymous method or lambda expression.

Delegates should not be confused with closures and inline functions. The concepts are related because a reference to a closure/inline function must be captured in a delegate reference to be useful at all. But a delegate does not always reference an inline function; it can also reference existing static or instance methods. Delegates form the basis of C# events, but should not be confused with those either.

Delegates were deliberately left out of Java because they were considered unnecessary and detrimental to the language, and because of potential performance issues.[27] Instead, alternative mechanisms are used. The wrapper pattern, which resembles the delegates of C# in that it allows the client to access one or more client-defined methods through a known interface, is one such mechanism. Another is the use of adapter objects using inner classes, which the designers of Java argued are a better solution than bound method references.[27]

See also example #C# delegates and equivalent Java constructs.

Lifted (nullable) types

C# allows value/primitive/simple types to be "lifted" to allow the special `null` value in addition to the type's native values. A type is lifted by adding a `?` suffix to the type name, this is equivalent to using the `Nullable<T>` generic type, where `T` is the type to be lifted. Conversions are implicitly defined to convert between values of the base and the lifted type. The lifted type can be compared against `null` or it can be tested for `HasValue`. Also, lifted operators are implicitly and automatically defined based on their non-lifted base, where — with the exception of some boolean operators — a null argument will propagate to the result.

Java does not support type lifting as a concept, but all of the built-in primitive types have corresponding wrapper types, which do support the `null` value by virtue of being reference types (classes).

According to the Java spec, any attempt to dereference the `null` reference must result in an exception being thrown at run-time, specifically a `NullPointerException`. (It would not make sense to dereference it otherwise, because, by definition, it points to no object in memory.) This also applies when attempting to unbox a variable of a wrapper type, which evaluates to `null`: the program will throw an exception, because there is no object to be unboxed - and thus no boxed value to take part in the subsequent computation.

The following example illustrates the different behavior. In C#, the lifted*operator propagates the `null` value of the operand; in Java, unboxing the null reference throws an exception.

Not all C# lifted operators have been defined to propagate `null` unconditionally, if one of the operands is `null`. Specifically, the boolean operators have been lifted to support ternary logic thus keeping impedance with SQL.

The Java boolean operators do not support ternary logic, nor is it implemented in the base class library.

Late-bound (dynamic) type

C# features a late bound dynamic type that supports no-reflection dynamic invocation, interoperability with dynamic languages, and ad-hoc binding to (for example) document object models. The `dynamic` type resolves member access dynamically at runtime as opposed to statically/virtual at compile time. The member lookup mechanism is extensible with traditional reflection as a fall-back mechanism.

There are several use cases for the `dynamic` type in C#:

- Less verbose use of reflection: By casting an instance to the `dynamic` type, members such as properties, methods, events etc. can be directly invoked on the instance without using the reflection API directly.
- Interoperability with dynamic languages: The dynamic type comes with a hub-and-spoke support for implementing dynamically typed objects and common runtime infrastructure for efficient member lookup.
- Creating dynamic abstractions on the fly: For instance, a dynamic object could provide simpler access to document object models such as XML or XHTML documents.

Java does not support a late-bound type. The use cases for C# dynamic type have different corresponding constructs in Java:

- For dynamic late-bound *by-name* invocation of preexisting types, reflection should be used.
- For interoperability with dynamic languages, some form of interoperability API specific to that language must be used. The Java virtual machine platform does have multiple dynamic languages implemented on it, but there is no common standard for how to pass objects between languages. Usually this involves some form of reflection or reflection-like API. As an example of how to use JavaFX objects from Java.[28]
- For creating and interacting with objects entirely at runtime, e.g., interaction with a document object model abstraction, a specific abstraction API must be used.

See also example #Interoperability with dynamic languages.

Pointers

Java precludes pointers and pointer-arithmetic within the Java runtime environment. The Java language designers reasoned that pointers are one of the main features that enable programmers to put bugs in their code and chose not to support them.[9] Java does not allow for directly passing and receiving objects/structures to/from the underlying operating system and thus does not need to model objects/structures to such a specific memory layout, layouts that frequently would involve pointers. Java's communication with the underlying operating system is instead based upon Java Native Interface (JNI) where communication with/adaptation to an underlying operating system is handled through an external *glue* layer.

While C# does allow use of pointers and corresponding pointer arithmetic, the C# language designers had the same concerns that pointers could potentially be used to bypass the strict rules for object access. Thus, C# by default also precludes pointers.[29] However, because pointers are needed when calling many native functions, pointers are allowed in an explicit *unsafe* mode. Code blocks or methods that use the pointers must be marked with the `unsafe` keyword to be able to use pointers, and the compiler requires the `/unsafe` switch to allow compiling such code. Assemblies that are compiled using the `/unsafe` switch are marked as such and may only execute if explicitly trusted. This allows using pointers and pointer arithmetic to directly pass and receive objects to/from the operating system or other native APIs using the native memory layout for those objects while also isolating such potentially unsafe code in specifically trusted assemblies.

Reference types

In both languages references are a central concept. All instances of classes are *by reference*.

While not directly evident in the language syntax *per se*, both languages support the concept of *weak* references. An instance that is only referenced by weak references is eligible for garbage collection just as if there were no references at all. In both languages this feature is exposed through the associated libraries, even though it is really a core runtime feature.

Along with weak references, Java has *soft references*. They are much like weak references, but the JVM will not deallocate softly-referenced objects until the memory is needed.

Reference types	Java	C#
Garbage collection	Yes	Yes
Weak references	Yes	Yes
Reference queue (interaction with garbage collection)	Yes	Yes
Soft references	Yes	Yes
Phantom references	Yes	No
Proxy support	Yes; proxy generation	Yes; object contexts

Arrays and collections

Arrays and collections are concepts featured by both languages.

Arrays and Collections	Java	C#
Abstract data types	Yes	Yes
One-dimensional, zero-based index arrays	Yes	Yes
Multidimensional arrays, rectangular (single array)	No	Yes
Multidimensional arrays, jagged (arrays of arrays)	Yes	Yes
Non-zero based arrays	No	Some
Unified arrays and collections	No	Yes
Maps/dictionaries	Yes	Yes
Sorted dictionaries	Yes	Yes[30]
Sets	Yes	Yes
Sorted sets	Yes	Yes[31]
Lists/vectors	Yes	Yes
Queues/stacks	Yes	Yes
Priority queue	Yes	Yes[32][33]
Bags/multisets	Third-party library	Yes
Concurrency optimized collections	Yes	Yes[34]

The syntax used to declare and access arrays is identical, except that C# has added syntax for declaring and manipulating multidimensional arrays.

Java	C#
Arrays are implicitly direct specializations of `Object`. They are not unified with collection types.	Arrays in C# are implicit specializations of the `System.Array` class that implements several collection interfaces.
Arrays and collections are completely separate with no unification. Arrays cannot be passed where sequences or collections are expected (though they can be wrapped using `Arrays.asList`).	Arrays can be passed where sequences (`IEnumerables`) or collections/list interfaces are expected. However, the collection operations that alter the number of elements (insert/add/remove) will throw exceptions as these operations are unsupported by arrays.
The `for` statement accepts either arrays or `Iterables`. All collections implement `Iterable`. This means that the same short syntax can be used in for-loops.	The `foreach` statement iterates through a sequence using the `IEnumerable` or `IEnumerable<T>` interface. Because arrays always implicitly implement these interfaces, the loop will iterate through arrays also.
In both languages arrays of reference types are covariant. This means that a `String[]` array is assignable to variables of `Object[]`, as `String` is a specialization of (assignable to) `Object`. In both languages, the arrays will perform a type check when inserting new values, because type safety would otherwise be compromised. This is in contrast to how generic collections have been implemented in both languages.	
No multidimensional arrays (rectangular arrays), but arrays of references to arrays (jagged arrays).	Multidimensional arrays (rectangular arrays), and arrays of references to arrays (jagged arrays).
Arrays cannot be resized (though use of the `System.arraycopy()` method can allow for multi-step array resizing)	Arrays can be resized while preserving existing values using the `Array.Resize()` static array method (but this may return a new array).
Implemented as a retrofit for the `java.util` library having extra features, like data structures like sets and linked sets, and has several algorithms to manipulate elements of a collection, like finding the largest element based on some `Comparator<T>` object, finding the smallest element, finding sublists within a list, reverse the contents of a list, shuffle the contents of a list, create immutable versions of a collection, performs sorts, and make binary searches.[35]	The C# collections framework consists of classes from the `System.Collections` and the `System.Collections.Generic` namespaces with several useful interfaces, abstract classes, and data structures.[36] NET 3.5 added `System.Linq` namespace that contains various extension methods for querying collections, such as `Aggregate`, `All`, `Average`, `Distinct`, `Join`, `Union` and many others. Queries using these methods are called Language Integrated Query (LINQ).

Multidimensional arrays can in some cases increase performance because of increased locality (as there is one pointer dereference instead of one for every dimension of the array, as it is the case for jagged arrays). However, since all array element access in a multidimensional array requires multiplication/shift between the two or more dimensions, this is an advantage only in very random access scenarios.

Another difference is that the entire multidimensional array can be allocated with a single application of operator `new`, while jagged arrays require loops and allocations for every dimension. Note, though, that Java provides a syntactic construct for allocating a jagged array with regular lengths; the loops and multiple allocations are then performed by the virtual machine and need not be explicit at the source level.

Both languages feature an extensive set of collection types that includes various ordered and unordered types of lists, maps/dictionaries, sets, etc.

Java also supports the syntax of C/C++:[37]

Java	C#

`// Is valid, as numbers is an object of type short[]`
`short[] numbers = new short[100];`
 	
`// Is valid, but it isn't clear code`
`double values[] = new double[100];`
	`// Is valid, as numbers is an object of type short[]`
	`short[] numbers = new short[100];`
	
	`// Won't compile!`
	`double values[] = new double[100];`

Expressions and operators

Expressions and operators	Java	C#
Arithmetic operators	Yes	Yes
Logical operators	Yes	Yes
Bitwise logic operators	Yes	Yes
Conditional	Yes	Yes
String concatenation	Yes	Yes
Casts	Yes	Yes
Boxing	Yes; implicit	Yes; implicit
Unboxing	Yes; implicit	Yes; explicit
Lifted operators	No, but see java.util.Optional	Yes
Overflow control	No	Yes
Strict floating point evaluation	Yes; opt-in/out	Yes; opt-in[38]
Verbatim (here-)strings	No	Yes[39]

Boxing and unboxing

Both languages allow automatic boxing and unboxing, i.e. they allow for implicit casting between any primitive types and the corresponding reference types.
In C#, the primitive types are subtypes of the Object type. In Java this is not true; any given primitive type and the corresponding wrapper type have no specific relationship with each other, except for autoboxing and unboxing, which act as syntactic sugar for interchanging between them. This was done intentionally, to maintain backward compatibility with prior versions of Java, in which no automatic casting was allowed, and the programmer worked with two separate sets of types: the primitive types, and the wrapper (reference) type hierarchy.[40]

This difference has the following consequences. First of all, in C#, primitive types can define methods, such as an override of Object's `ToString()` method. In Java, this task is accomplished by the primitive wrapper classes.
Secondly, in Java an extra cast is needed whenever one tries to directly dereference a primitive value, as it will not be boxed automatically. The expression `((Integer)42).toString()` will convert an integer literal to string in Java while `42.ToString()` performs the same operation in C#. This is because the latter one is an instance call on the primitive value `42`, while the former one is an instance call on an object of type `java.lang.Integer`.

Finally, another difference is that Java makes heavy use of boxed types in generics (see below).

Statements

Statements	Java	C#
Loops	Yes	Yes
Conditionals	Yes	Yes
Flow control	Yes	Yes
Assignment	Yes	Yes
Exception control	Yes	Yes
Variable declaration	Yes	Yes
Variable type inference	Third-party library[41]	Yes
Deterministic disposal (ARM-blocks)	Yes	Yes

Syntax

Both languages are considered "curly brace" languages in the C/C++ family. Overall the syntaxes of the languages are very similar. The syntax at the statement and expression level is almost identical with obvious inspiration from the C/C++ tradition. At type definition level (classes and interfaces) some minor differences exist. Java is explicit about extending classes and implementing interfaces, while C# infers this from the kind of types a new class/interface derives from.

C# supports more features than Java, which to some extent is also evident in the syntax that specifies more keywords and more grammar rules than Java.

Keywords and backward compatibility

As the languages evolved, the language designers for both languages have faced situations where they wanted to extend the languages with new keywords or syntax. New keywords in particular may break existing code at source level, i.e. older code may no longer compile, if presented to a compiler for a later version of the language. Language designers are keen to avoid such regressions. The designers of the two languages have been following different paths when addressing this problem.

Java language designers have avoided new keywords as much as possible, preferring instead to introduce new syntactic constructs that were not legal before or to reuse existing keywords in new contexts. This way they didn't jeopardize backward compatibility. An example of the former can be found in how the `for` loop was extended to accept iterable types. An example of the latter can be found in how the `extends` and (especially) the `super` keywords were reused for specifying type bounds when generics were introduced in Java 1.5. At one time (Java 1.4) a new keyword `assert` was introduced that was not reserved as a keyword before. This had the potential to render previously valid code invalid, if for instance the code used `assert` as an identifier. The designers chose to address this problem with a four-step solution: 1) Introducing a compiler switch that indicates if Java 1.4 or later should be used, 2) Only marking `assert` as a keyword when compiling as Java 1.4 and later, 3) Defaulting to 1.3 to avoid rendering previous (non 1.4 aware code) invalid and 4) Issue warnings, if the keyword is used in Java 1.3 mode, in order to allow the developers to change the code.

C# language designers have introduced several new keywords since the first version. However, instead of defining these keywords as *global* keywords, they define them as *context sensitive* keywords. This means that even when they introduced (among others) the `partial` and `yield` keywords in C# 2.0, the use of those words as identifiers is still valid as there is no clash possible between the use as keyword and the use as identifier, given the context. Thus, the present C# syntax is fully backward compatible with source code written for any previous version without specifying the language version to be used.

keyword	feature, example usage
`checked`, `unchecked`	In C#, `checked` statement blocks or expressions can enable run-time checking for arithmetic overflow.[42]
`get`, `set`	C# implements properties as part of the language syntax with their optional corresponding `get` and `set` accessors, as an alternative for the accessor methods used in Java, which is not a language feature but a coding-pattern based on method name conventions.

`goto`	C# supports the `goto` keyword. This can occasionally be useful, for example for implementing finite state machines or for generated code, but the use of a more structured method of control flow is usually recommended (see criticism of the goto statement). Java does not support the `goto` statement (but `goto` is a reserved word). However, Java does support labeled `break` and `continue` statements, which in certain situations can be used when a `goto` statement might otherwise be used.
	`switch(color)`
	`{`
	`  case Color.Blue:`
	`  Console.WriteLine("Color is blue");`
	`  break;`
	`  case Color.DarkBlue:`
	`  Console.WriteLine("Color is dark");`
	`  goto case Color.Blue;`
	`  // ...`
	`}`

`lock`	In C#, the `lock` keyword is a shorthand for synchronizing access to a block of code across threads (using a `Monitor`), wrapped in a `try` … `finally` block.
`out`, `ref`	C# has support for output and reference parameters. These allow returning multiple output values from a method, or passing values by reference.
`strictfp`	Java uses `strictfp` to guarantee the results of floating point operations remain the same across platforms.
`switch`	In C#, the switch statement also operates on strings and longs. Fallthrough is allowed for empty statements and possible via 'goto case' for statements containing code. Java's switch statement operates on strings (since Java 7) but not the `long` primitive type, and falls through for all statements (excluding those with '`break`').[43]
`synchronized`	In Java, the `synchronized` keyword is a shorthand for synchronizing access to a block of code across threads (using a `Monitor`), wrapped in a `try` … `finally` block.

`throws`	Java requires every method to declare the checked exceptions or superclasses of the checked exceptions that it can throw. Any method can also optionally declare the unchecked exception that it throws. C# has no such syntax.
	`public int readItem() throws java.io.IOException {`
	`  // ...`
	`}`

`using`	In C#, `using` causes the `Dispose` method (implemented via the `IDisposable` interface) of the object declared to be executed after the code block has run or when an exception is thrown within the code block.
	`// Create a small file "test.txt", write a string,`
	`// ... and close it (even if an exception occurs)`
	`using (StreamWriter file = new StreamWriter("test.txt"))`
	`{`
	`  file.Write("test");`
	`}`
	In Java SE 7 a similar construct has been added[44] called try-with-resources:
	`try (BufferedReader br = new BufferedReader(new FileReader(path))) {`
	`  return br.readLine();`
	`}`

Object-oriented programming

Both C# and Java are designed from the ground up as object-oriented languages using dynamic dispatch, with syntax similar to C++ (C++ in turn derives from C). Neither language is a superset of C or C++, however.

Object orientation	Java	C#
Classes	mandatory	mandatory
Interfaces	Yes	Yes
Abstract classes	Yes	Yes
Member accessibility levels	Yes; public, package, protected, private	Yes; public, internal, protected, private, protected internal
Class-level inner classes	Yes;`static` inner classes are class level	Yes; all inner classes are class level
Instance-level inner classes	Yes	No
Statement-level (local) anonymous classes	Yes	Yes; but without methods
Partial classes	No; Third-party library[45]	Yes
Implicit (inferred) anonymous classes	No	Yes[46]
Deprecation/obsolescence	Yes	Yes
Overload versioning	Some	Yes
Enums can implement interfaces	Yes	No
Properties	No, but see JavaBeans spec	Yes
Events	Provided by standard libraries	Built-in language feature
Operator overloading	No	Yes
Indexers	No	Yes
Implicit conversions	No; but see autoboxing (https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html)	Yes
Explicit conversions	Yes	Yes

Partial class

C# allows a class definition to be split across several source files using a feature called *partial classes*. Each part must be marked with the keyword `partial`. All the parts must be presented to the compiler as part of a single compilation. Parts can reference members from other parts. Parts can implement interfaces and one part can define a base class. The feature is useful in code generation scenarios (such as user interface (UI) design), where a code generator can supply one part and the developer another part to be compiled together. The developer can thus edit their part without the risk of a code generator overwriting that code at some later time. Unlike the class extension mechanism, a partial class allows *circular* dependencies among its parts as they are guaranteed to be resolved at compile time. Java has no corresponding concept.

Inner and local classes

Both languages allow *inner classes*, where a class is defined lexically inside another class. However, in each language these inner classes have rather different semantics.

In Java, unless the inner class is declared `static`, a reference to an instance of an inner class carries a reference to the outer class with it. As a result, code in the inner class has access to both the static and non-static members of the outer class. To create an instance of a non-static inner class, the instance of the embracing outer class must be named.[47] This is done via a new `new`-operator introduced in JDK 1.3: `outerClassInstance.new Outer.InnerClass()`. This can be done in any class that has a reference to an instance of the outer class.

In C#, an inner class is conceptually the same as a normal class. In a sense, the outer class only acts as a namespace. Thus, code in the inner class cannot access non-static members of the outer class unless it does so through an explicit reference to an instance of the outer class. Programmers can declare the inner class *private* to allow only the outer class to have any access to it.

Java provides another feature called *local classes* or *anonymous classes*, which can be defined within a method body. These are generally used to implement an interface with only one or two methods, which are typically event handlers. However, they can also be used to override virtual methods of a superclass. The methods in those local classes have access to the outer method's local variables declared `final`. C# satisfies the use-cases for these by providing anonymous delegates; see event handling for more about this.

C# also provides a feature called *anonymous types/classes*, but it is rather different from Java's concept with the same name. It allows the programmer to instantiate a class by providing only a set of names for the properties the class should have, and an expression to initialize each. The types of the properties are inferred from the types of those expressions. These implicitly-declared classes are derived directly from *object*.

Event

C# multicast-delegates are used with *events*. Events provide support for event-driven programming and is an implementation of the observer pattern. To support this there is a specific syntax to define events in classes, and operators to register, unregister or combine event handlers.

See here for information about how events are implemented in Java.

Operator overloading and conversions

Operator overloading and user-defined casts are separate features that both aim to allow new types to become first-class citizens in the type system. By using these features in C#, types such as `Complex` and `decimal` have been integrated so that the usual operators like addition and multiplication work with the new types. Unlike C++, C# does restrict the use of operator overloading, prohibiting it for the operators `new`, `( )`, `||`, &&, =, and any variations of compound statements like `+=`. But compound operators will call overloaded simple operators, like `-=` calling `-` and `=`.[48]

Java does not include operator overloading, nor custom conversions in order to prevent abuse of the feature and to keep the language simple.[49]

Indexer

C# also includes *indexers* that can be considered a special case of operator overloading (like the C++ `operator[]`), or parameterized `get`/`set` properties. An indexer is a property named `this[]` that uses one or more parameters (indexes); the indices can be objects of any type:

```
myList[4] = 5;
string name = xmlNode.Attributes["name"];
orders = customerMap[theCustomer];
```

Java does not include indexers. The common Java pattern involves writing explicit getters and setters where a C# programmer would use an indexer.

Fields and initialization

Fields and initialization	Java	C#
Fields	Yes	Yes
Constants	Yes	Yes; but no support for constant passed parameters[50]
Static (class) constructors	Yes	Yes
Instance constructors	Yes	Yes
Finalizers/destructors	Yes	Yes
Instance initializers	Yes	No; can be simulated with instance constructor
Object initialization	Bottom-up (fields and constructors)	Top-down (fields); bottom-up (constructors)
Object initializers	Yes	Yes
Collection initializers	No; can be simulated with methods initializers	Yes
Array initializers	Yes	Yes

Object initialization

In both C# and Java, an object's fields can be initialized either by *variable initializers* (expressions that can be assigned to variables where they are defined) or by *constructors* (special subroutines that are executed when an object is being created). In addition, Java contains *instance initializers*, which are anonymous blocks of code with no arguments that are run after the explicit (or implicit) call to a superclass's constructor but before the constructor is executed.

C# initializes object fields in the following order when creating an object:

1. Derived static fields
2. Derived static constructor
3. Derived instance fields
4. Base static fields
5. Base static constructor
6. Base instance fields
7. Base instance constructor
8. Derived instance constructor

Some of the above fields may not be applicable (e.g. if an object does not have *static fields*). *Derived fields* are those that are defined in the object's direct class, while *base field* is a term for the fields that are defined in one of the object's superclasses. Note that an object representation in memory contains all fields defined in its class or any of its superclasses, even, if some fields in superclasses are defined as private.

It is guaranteed that any field initializers take effect before any constructors are called, since both the instance constructor of the object's class and its superclasses are called after field initializers are called. There is, however, a potential trap in object initialization when a virtual method is called from a base constructor. The overridden method in a subclass may reference a field that is defined in the subclass, but this field may not have been initialized because the constructor of the subclass that contains field initialization is called after the constructor of its base class.

In Java, the order of initialization is as follows:

1. Invocation of another constructor (either of the object's class or of the object's superclass)
2. Instance variable initializers and instance initializers (in the order they appear in the source code)
3. The constructor body

Like in C#, a new object is created by calling a specific constructor. Within a constructor, the first statement may be an invocation of another constructor. If this is omitted, the call to the argumentless constructor of the superclass is added implicitly by the compiler. Otherwise, either another overloaded constructor of the object's class can be called explicitly, or a superclass constructor can be called. In the former case, the called constructor will again call another constructor (either of the object's class or its subclass) and the chain sooner or later ends up at the call to one of the constructors of the superclass.

After another constructor is called (that causes direct invocation of the superclass constructor, and so forth, down to the Object class), instance variables defined in the object's class are initialized. Even if there are no variable initializers explicitly defined for some variables, these variables are initialized to default values. Note that instance variables defined in superclasses are already initialized by this point, because they were initialized by a superclass constructor when it was called (either by the constructor's code or by variable initializers performed before the constructor's code or implicitly to default values). In Java, variable initializers are executed according to their textual order in the source file.

Finally, the constructor body is executed. This ensures proper order of initialization, i.e. the fields of a base class finish initialization before initialization of the fields of an object class begins.

There are two main potential traps in Java's object initialization. First, variable initializers are expressions that can contain method calls. Since methods can reference any variable defined in the class, the method called in a variable initializer can reference a variable that is defined below the variable being initialized. Since initialization order corresponds to textual order of variable definitions, such a variable would not be initialized to the value prescribed by its initializer and would contain the default value. Another potential trap is when a method that is overridden in the derived class is called in the base class constructor, which can lead to behavior the programmer would not expect when an object of the derived class is created. According to the initialization order, the body of the base class constructor is executed before variable initializers are evaluated and before the body of the derived class constructor is executed. The overridden method called from the base class constructor can, however, reference variables defined in the derived class, but these are not yet initialized to the values specified by their initializers or set in the derived class constructor. The latter issue applies to C# as well, but in a less critical form since in C# methods are not overridable by default.

Resource disposal

Both languages mainly use garbage collection as a means of reclaiming memory resources, rather than explicit deallocation of memory. In both cases, if an object holds resources of different kinds other than memory, such as file handles, graphical resources, etc., then it must be notified explicitly when the application no longer uses it. Both C# and Java offer interfaces for such deterministic disposal and both C# and Java (since Java 7) feature automatic resource management statements that will automatically invoke the disposal/close methods on those interfaces.

Methods

Methods and properties	Java	C#
Static imports	Yes	Yes[51]
Virtual methods	Virtual by default	Non-Virtual by default
Abstract	Yes	Yes
Sealing	Yes	Yes
Explicit interface implementation	Default methods	Yes[52]
Value (input) parameters	Yes	Yes
Reference (input/output) parameters	No	Yes
Output (output) parameters	No	Yes
Constant (immutable) parameters	Yes; final parameters	No
Variadic methods	Yes	Yes
Optional arguments	No;[53] Can be simulated with method overloading or varargs	Yes
Named arguments	No	Yes
Generator methods	No	Yes
Extension/default methods	Yes	Yes
Conditional methods	No	Yes
Partial methods	No	Yes

Extension methods and default methods

Using a special *this* designator on the first parameter of a method, C# allows the method to act as if it were a member method of the type of the first parameter. This *extension* of the foreign class is purely syntactical. The extension method must be declared `static` and defined within a purely static class. The method must obey any member access restriction like any other method external to the class; thus static methods cannot break object encapsulation.[54][55] The "extension" is only active within scopes where the namespace of the static host class has been imported.

Since Java 8, Java has a similar feature called *default methods*, which are methods with a body declared on interfaces. As opposed to C# extension methods, Java default methods are instance methods on the interface that declare them. Definition of default methods in classes that implement the interface is optional: If the class does not define the method, the default definition is used instead.

Both the C# extension methods and the Java default methods allow a class to override the default implementation of the extension/default method, respectively. In both languages this override is achieved by defining a method on the class that should use an alternate implementation of the method.

C# scope rules defines that if a matching method is found on a class, it takes precedence over a matching extension method. In Java any class declared to implement an interface with default method is assumed to have the default methods implementions, *unless* the class implements the method itself.

Partial methods

Related to *partial classes* C# allows partial methods to be specified within partial classes. A partial method is an intentional declaration of a method with several restrictions on the signature. The restrictions ensure that if a definition is not provided by any class part, then the method and every call to it can be safely erased.[56] This feature allows code to provide a large number of interception points (like the template method GoF design pattern) without paying any runtime overhead if these extension points are not being used by another class part at compile time. Java has no corresponding concept.

Virtual methods

Methods in C# are non-virtual by default, and must be declared virtual explicitly, if desired. In Java, all non-static non-private methods are virtual. Virtuality guarantees that the most recent override for the method will always be called, but incurs a certain runtime cost on invocation as these invocations cannot be normally inlined, and require an indirect call via the virtual method table. However, some JVM implementations, including the Oracle reference implementation, implement inlining of the most commonly called virtual methods.

Java methods are virtual by default (although they can be *sealed* by using the `final` modifier to disallow overriding). There is no way to let derived classes define a new, unrelated method with the same name.

This means that by default in Java, and only when explicitly enabled in C#, new methods may be defined in a derived class with the same name and signature as those in its base class. When the method is called on a superclass reference of such an object, the "deepest" overridden implementation of the base class' method will be called according to the specific subclass of the object being referenced.

In some cases, when a subclass introduces a method with the same name and signature as a method already present in the base class, problems can occur. In Java, this will mean that the method in the derived class will implicitly override the method in the base class, even though that may not be the intent of the designers of either class.

To mitigate this, C# requires that if a method is intended to override an inherited method, the `override` keyword must be specified. Otherwise, the method will "hide" the inherited method. If the keyword is absent, compiler warning to this effect is issued, which can be silenced by specifying the `new` keyword. This avoids the problem that can arise from a base class being extended with a non-private method (i.e. an inherited part of the namespace) whose signature is already in use by a derived class. Java has a similar compiler check in the form of the `@Override` method annotation, but it is not compulsory, and in its absence, most compilers will not provide comment (but the method will be overridden).

Constant/immutable parameters

In Java, it is possible to prevent reassignment of a local variable or method parameter by using the **`final`** keyword. Applying this keyword to a primitive type variable causes the variable to become immutable. However, applying **`final`** to a reference type variable only prevents that another object is assigned to it. It will not prevent the data contained by the object from being mutated. There is no C# equivalent.[50]

Java	C#

`public int addOne(final int x) {`
`  x++;  // ERROR: a final variable cannot be reassigned`
`  return x;`
`}`
 	
`public ArrayList addOne(final ArrayList list) {`
`  list.add(1);  // OK: it is still possible to modify a`
`  // final (reference type) variable`
`  return list;`
`}`
	no C# language equivalent

Both languages do not support essential feature of const-correctness that exists in C/C++, which makes a method constant.

Interestingly, Java defines the word "constant" arbitrarily as a **`static final`** field. Only these variables are capital-only variables, where the names are separated with an underscore. A parameter that is only **`final`** is not considered as a constant, although it may be so in the case of a primitive data type or an immutable class, like a `String`.

Generator methods

Any C# method declared as returning `IEnumerable`, `IEnumerator` or the generic versions of these interfaces can be implemented using `yield` syntax. This is a form of limited, compiler-generated continuations and can drastically reduce the code needed to traverse or generate sequences, although that code is just generated by the compiler instead. The feature can also be used to implement infinite sequences, e.g., the sequence of Fibonacci numbers.

Java does not have an equivalent feature. Instead generators are typically defined by providing a specialized implementation of a well-known collection or iterable interface, which will compute each element on demand. For such a generator to be used in a *for each* statement, it must implement interface `java.lang.Iterable`.

See also example Fibonacci sequence below.

Explicit interface implementation

C# also has *explicit interface implementation* that allows a class to specifically implement methods of an interface, separate to its own class methods, or to provide different implementations for two methods with the same name and signature inherited from two base interfaces.

In either language, if a method (or property in C#) is specified with the same name and signature in multiple interfaces, the members will clash when a class is designed that implements those interfaces. An implementation will by default implement a common method for all of the interfaces. If separate implementations are needed (because the methods serve separate purposes, or because return values differ between the interfaces) C#'s explicit interface implementation will solve the problem, though allowing different results for the same method, depending on the current cast of the object. In Java there is no way to solve this problem other than refactoring one or more of the interfaces to avoid name clashes.[52]

Reference (in/out) parameters

The arguments of primitive types (e.g. int, double) to a method are passed by value in Java whereas objects are passed by reference. This means that a method operates on copies of the primitives passed to it instead of on the actual variables. On the contrary, the actual objects in some cases can be changed. In the following example object String is not changed. Object of class 'a' is changed.

In C#, it is possible to enforce a reference with the `ref` keyword, similar to C++ and in a sense to C. This feature of C# is particularly useful when one wants to create a method that returns more than one object. In Java trying to return multiple values from a method is unsupported, unless a wrapper is used, in this case named "Ref".[57]

Java

```
class PassByRefTest {

    static class Ref<R>{
        R val;
        Ref(R v) {val = v;}
    }

    public static void changeMe(Ref<String> s) {
        s.val = "Changed";
    }

    public static void swap(Ref<Integer> x, Ref<Integer> y) {
        int temp = x.val;

        x.val = y.val;
        y.val = temp;
    }

    public static void main(String[] args) {
        Ref<Integer> a =  new Ref(5);
        Ref<Integer> b = new Ref(10);
        Ref<String> s = new Ref("still unchanged");

        swap(a, b);
        changeMe(s);

        System.out.println( "a = " + a.val + ", " +
                            "b = " + b.val + ", " +
                            "s = " + s.val );
    }
}
```

```
a = 10, b = 5, s = Changed
```

C#

```
class PassByRefTest {

    public static void ChangeMe(out string s) {
        s = "Changed";
    }

    public static void Swap(ref int x, ref int y) {
        int temp = x;

        x = y;
        y = temp;
    }

    public static void Main(string[] args) {
        int a = 5, b = 10;
        string s = "still unchanged";

        Swap(ref a, ref b);
        ChangeMe(out s);

        System.Console.WriteLine("a = " + a + ", " +
                                 "b = " + b + ", " +
                                 "s = " + s);
    }
}
```

```
a = 10, b = 5, s = Changed
```

Exceptions

Exceptions	Java	C#
Checked exceptions	Yes	No
Try-catch-finally	Yes	Yes

Checked exceptions

Java supports checked exceptions (along with unchecked exceptions). C# only supports unchecked exceptions. Checked exceptions force the programmer to either declare the exception thrown in a method, or to catch the thrown exception using a `try-catch` clause.

Checked exceptions can encourage good programming practice, ensuring that all errors are dealt with. However Anders Hejlsberg, chief C# language architect, argues that they were to some extent an experiment in Java and that they have not been shown to be worthwhile except in small example programs.[58][59]

One criticism is that checked exceptions encourage programmers to use an empty catch block (`catch (Exception e) {}`),[60] which silently swallows exceptions, rather than letting the exceptions propagate to a higher-level exception-handling routine. In some cases, however, exception chaining can be applied instead, by re-throwing the exception in a wrapper exception. For example, if an object is changed to access a database instead of a file, an `SQLException (https://docs.oracle.com/javase/9/docs/api/java/sql/SQLException.html)` could be caught and re-thrown as an `IOException (https://docs.oracle.com/javase/9/docs/api/java/io/IOException.html)`, since the caller may not need to know the inner workings of the object.

However, not all programmers agree with this stance. James Gosling and others maintain that checked exceptions are useful, and misusing them has caused the problems. Silently catching exceptions is possible, yes, but it must be stated explicitly what to do with the exception, versus unchecked exceptions that allow doing nothing by default. It can be ignored, but code must be written explicitly to ignore it.[61][62]

Try-catch-finally

There are also differences between the two languages in treating the `try-finally` statement. The `finally` block is always executed, even if the `try` block contains control-passing statements like `throw` or `return`. In Java, this may result in unexpected behavior, if the `try` block is left by a `return` statement with some value, and then the `finally` block that is executed afterward is also left by a `return` statement with a different value. C# resolves this problem by prohibiting any control-passing statements like `return` or `break` in the `finally` block.

A common reason for using `try-finally` blocks is to guard resource managing code, thus guaranteeing the release of precious resources in the finally block. C# features the `using` statement as a syntactic shorthand for this common scenario, in which the `Dispose()` method of the object of the `using` is always called.

A rather subtle difference is the moment a stack trace is created when an exception is being thrown. In Java, the stack trace is created in the moment the exception is created.

```
class Foo {
    Exception up = new Exception();
    int foo() throws Exception {
        throw up;
    }
}
```

The exception in the statement above will always contain the constructor's stack-trace - no matter how often foo is called. In C# on the other hand, the stack-trace is created the moment "throw" is executed.

```
class Foo
{
    Exception e = new Exception();
    int foo()
    {
        try
        {
            throw e;
        }
        catch (Exception e)
        {
            throw;
        }
    }
}
```

In the code above, the exception will contain the stack-trace of the first throw-line. When catching an exception, there are two options in case the exception should be rethrown: `throw` will just rethrow the original exception with the original stack, while `throw e` would have created a new stack trace.

Finally blocks

Java allows flow of control to leave the `finally` block of a `try` statement, regardless of the way it was entered. This can cause another control flow statement (such as `return`) to be terminated mid-execution. For example:

```
int foo() {
    try {
        return 0;
    } finally {
        return 1;
    }
}
```

In the above code, the `return` statement within `try` block causes control to leave it, and thus `finally` block is executed before the actual return happens. However, the `finally` block itself also performs a return. Thus, the original return that caused it to be entered is not executed, and the above method returns 1 rather than 0. Informally speaking, it *tries* to return 0 but *finally* returns 1.

C# does not allow any statements that allow control flow to leave the `finally` block prematurely, except for `throw`. In particular, `return` is not allowed at all, `goto` is not allowed if the target label is outside the `finally` block, and `continue` and `break` are not allowed if the nearest enclosing loop is outside the `finally` block.

Generics

In the field of generics the two languages show a superficial syntactical similarity, but they have deep underlying differences.

Generics	Java	C#
Implementation	Type erasure	Reification
Runtime realization	No	Yes
Type variance	Use-site	Declaration-site (only on interfaces)
Reference type constraint	Yes; implicit	Yes
Value/primitive type constraint	No	Yes
Constructor constraint	No	Yes (only for parameterless constructor)
Subtype constraint	Yes	Yes
Supertype constraint	Yes	No
Migration compatibility	Yes	No

Type erasure versus reified generics

Generics in Java are a language-only construction; they are implemented only in the compiler. The generated classfiles include generic signatures only in form of metadata (allowing the compiler to compile new classes against them). The runtime has no knowledge of the generic type system; generics are not part of the JVM. Instead, generics classes and methods are transformed during compiling via a process termed type erasure. During this, the compiler replaces all generic types with their *raw* version and inserts casts/checks appropriately in client code where the type and its methods are used. The resulting byte code will contain no references to any generic types or parameters (See also Generics in Java).
The language specification intentionally prohibits certain uses of generics; this is necessary to allow for implementing generics through type erasure, and to allow for migration compatibility.[63]

C# builds on support for generics from the virtual execution system, i.e., it is not just a language feature. The language is merely a front-end for cross-language generics support in the CLR. During compiling generics are verified for correctness, but code generation to *implement* the generics are deferred to class-load time. Client code (code invoking generic methods/properties) are fully compiled and can safely assume generics to be type-safe. This is called reification. At runtime, when a unique set of type parameters for a generic class/method/delegate is encountered for the first time, the class loader/verifier will synthesize a concrete class descriptor and generate method implementations. During the generation of method implementations all reference types will be considered one type, as reference types can safely share the same implementations. This is merely for the purpose of *implementing* code. Different sets of reference types will still have unique type descriptors; their method tables will merely point to the same code.

The following list illustrates some differences between Java and C# when managing generics. It is not exhaustive:[64]

Java	C#
Type checks and downcasts are injected into client code (the code *referencing* the generics). Compared to non-generic code with manual casts, these casts will be the same,[65] but compared to compile-time verified code that would not need runtime casts and checks, these operations represent a performance overhead.	C#/.NET generics guarantee type-safety and are verified at compile time, making extra checks/casts are unnecessary at runtime. Hence, generic code will run *faster* than non-generic (or type-erased) code that require casts when handling non-generic or type-erased objects.
Cannot use primitive types as type parameters; instead, the developer must use the wrapper type corresponding to the primitive type. This incurs extra performance overhead by requiring boxing and unboxing conversions as well a memory and garbage collection pressure, as the wrappers will be heap-allocated as opposed to stack-allocated.	Primitive and value types are allowed as type parameters in generic realizations. At runtime code will be synthesized and compiled for each unique combination of type parameters upon first use. Generics that are realized with primitive/value type do not require boxing/unboxing conversions.
Generic exceptions are not allowed[66] and a type parameter cannot be used in a catch clause[67]	Can both define generic exceptions and use those in catch clauses
Static members are shared across all generic realizations[68] (during type erasure all realizations are folded into a single class)	Static members are separate for each generic realization. A generic realization is a unique class.
Type parameters cannot be used in declarations of static fields/methods or in definitions of static inner classes	No restrictions on use of type parameters

Cannot create an array where the component type is a generic realization (concrete parameterized type)
`Object tenPairs = new Pair<Integer, String>[10]; // error`
`Pair<String, String>[] tenPairs = new Pair[10]; //OK`
	A generic realization is a 1st class citizen and can be used as any other class; also an array component
	`object tenPairs = new Pair<int, string>[10]; // OK`

Cannot create an array where the component type is a type parameter, but it is valid to create an `Object` array and perform a typecast on the new array to achieve the same effect.
`public class Lookup<K, V> {`
`  public V[] getEmptyValues(K key, Class<V> vType) {`
`  return (V[]) new Object[0]; // OK`
`  }`
`}`
When a generic type parameter is under inheritance constraints the constraint type may be used instead of `Object`
`public class Lookup<K, V extends Comparable<V>> {`
`  public V[] getEmptyValues(K key) {`
`  return (V[]) new Comparable[0];`
`  }`
`}`
	Type parameters represent actual, discrete classes and can be used like any other type within the generic definition.
	`public class Lookup<K, V> {`
	`  public V[] GetEmptyValues(K key) {`
	`  return new V[0]; // OK`
	`  }`
	`}`

There is no class literal for a concrete realization of a generic type	A generic realization is an actual class.
`instanceof` is not allowed with type parameters or concrete generic realizations	The `is` and `as` operators work the same for type parameters as for any other type.
Cannot create new instances using a type parameter as the type	With a constructor constraint, generic methods or methods of generic classes can create instances of classes that have default constructors.
Type information is erased during compiling. Special extensions to reflection must be used to discover the original type.	Type information about C# generic types is fully preserved at runtime, and allows full reflection support and instantiation of generic types.
Reflection cannot be used to construct new generic realizations. During compilation extra code (typecasts) are injected into the *client* code of generics. This precludes creating new realizations later.	Reflection can be used to create new realizations for new combinations of type parameters.

C# allows generics directly for primitive types. Java, instead, allows the use of boxed types as type parameters (e.g., `List<Integer>` instead of `List<int>`). This comes at a cost since all such values need to be boxed/unboxed when used, and they all need to be heap-allocated. However, a generic type can be specialized with an array type of a primitive type in Java, for example `List<int[]>` is allowed.[69] Several third-party libraries implemented the basic collections in Java with backing primitive arrays to preserve the runtime and memory optimization that primitive types provide.[70]

Migration compatibility

Java's type erasure design was motivated by a design requirement to achieve *migration compatibility* - not to be confused with backward compatibility. In particular, the original requirement was "… *there should be a clean, demonstrable migration path for the Collections APIs that were introduced in the Java 2 platform*".[40] This was designed so that any new generic collections should be passable to methods that expected one of the pre-existing collection classes.[71]

C# generics were introduced into the language while preserving full backward compatibility, but did not preserve full *migration compatibility*: Old code (pre C# 2.0) runs unchanged on the new generics-aware runtime without recompilation. As for *migration compatibility*, new generic collection classes and interfaces were developed that supplemented the non-generic .NET 1.x collections rather than replacing them. In addition to generic collection interfaces, the new generic collection classes implement the non-generic collection interfaces where possible. This prevents the use of new generic collections with pre-existing (non-generic aware) methods, if those methods are coded to use the collection *classes*.

Covariance and contravariance

Covariance and contravariance is supported by both languages. Java has use-site variance that allows a single generic class to declare members using both co- and contravariance. C# has define-site variance for generic interfaces and delegates. Variance is unsupported directly on classes but is supported through their implementation of variant interfaces. C# also has use-site covariance support for methods and delegates.

Functional programming

Functional programming	Java	C#
Method references	Yes[10]	Yes
Closures	All lambdas do not introduce a new level of scope. All referenced variables must be effectively final	Yes
Lambda expressions	Yes[72]	Yes
Expression trees	No	Yes
Generic query language	No; but see 'Java 8 Stream equivalent' (Monad)[73]	Yes
Tail recursion compiler optimizations	No	Only on x64[74]

Closures

A closure is an inline function that captures variables from its lexical scope.

C# supports closures as anonymous methods or lambda expressions with full-featured closure semantics.[75][76]

In Java, anonymous inner classes will remain the preferred way to emulate closures until Java 8 has become the new standard. This is a more verbose construction. This approach also has some differences compared to real closures, notably more controlled access to variables from the enclosing scopes: only final members can be referenced. Java 8, however introduces lambdas that fully inherit the current scope and, in fact, do not introduce a new scope.

When a reference to a method can be passed around for later execution, a problem arises about what to do when the method has references to variables/parameters in its lexical scope. C# closures can access any variable/parameter from its lexical scope. In Java's anonymous inner classes, only references to final members of the lexical scope are allowed, thus requiring the developer to mark which variables to make available, and in what state (possibly requiring boxing).

Lambdas and expression trees

C# and Java feature a special type of in-line closures called lambdas. These are anonymous methods: they have a signature and a body, but no name. They are mainly used to specify local function-valued arguments in calls to other methods, a technique mainly associated with functional programming.

C#, unlike Java, allows the use of lambda functions as a way to define special data structures called expression trees. Whether they are seen as an executable function or as a data structure depends on compiler type inference and what type of variable or parameter they are assigned or cast to. Lambdas and expression trees play key roles in Language Integrated Query (LINQ).

Metadata

Metadata	Java	C#
Metadata annotations/attributes	Interface based; user-defined annotations can be created[77]	Class based
Positional arguments	No; unless a single argument	Yes
Named arguments	Yes	Yes
Default values	At definition	Through initialization
Nested types	Yes	Yes
Specialization	No	Yes
Conditional metadata	No	Yes

Preprocessing, compilation and packaging

Preprocessing, Compilation and Packaging	Java	C#
Namespaces	Packages	Namespaces
File contents	Restricted	Free
Packaging	Package	Assembly
Classes/assembly search path	ClassPath	Both compile-time and runtime[78][79]
Conditional compilation	No; but see Apache Ant[80]	Yes
Custom errors/warnings	Yes; AnnotationProcessor	Yes
Explicit regions	No	Yes

Namespaces and file contents

In C#, namespaces are similar to those in C++. Unlike package names in Java, a namespace is not in any way tied to the location of the source file. While it is not strictly necessary for a Java source file location to mirror its package directory structure, it is the conventional organization.

Both languages allow importing of classes (e.g., **`import java.util.*`** in Java), allowing a class to be referenced using only its name. Sometimes classes with the same name exist in multiple namespaces or packages. Such classes can be referenced by using fully qualified names, or by importing only selected classes with different names. To do this, Java allows importing a single class (e.g., **`import java.util.List`**). C# allows importing classes under a new local name using the following syntax: **`using`** `Console = System.Console`. It also allows importing specializations of classes in the form of **`using`** `IntList = System.Collections.Generic.List<`**`int`**`>`.

Java has a static import syntax that allows using the short name of some or all of the static methods/fields in a class (e.g., allowing `foo(bar)` where `foo()` can be statically imported from another class). C# has a static class syntax (not to be confused with static inner classes in Java), which restricts a class to only contain static methods. C# 3.0 introduces extension methods to allow users to statically add a method to a type (e.g., allowing `foo.bar()` where `bar()` can be an imported extension method working on the type of `foo`).

The Sun Microsystems Java compiler requires that a source file name must match the only public class inside it, while C# allows multiple public classes in the same file, and puts no restrictions on the file name. C# 2.0 and later allows splitting a class definition into several files by using the `partial` keyword in the source code. In Java, a public class will always be in its own source file. In C#, source code files and logical units separation are not tightly related.

Conditional compilation

Unlike Java, C# implements conditional compilation using preprocessor directives. It also provides a `Conditional` attribute to define methods that are only called when a given compilation constant is defined. This way, assertions can be provided as a framework feature with the method `Debug.Assert()`, which is only evaluated when the `DEBUG` constant is defined. Since version 1.4, Java provides a language feature for assertions, which are turned off at runtime by default but can be enabled using the `-enableassertions` or `-ea` switch when invoking the JVM.

Threading and asynchronous features

Both languages include thread synchronization mechanisms as part of their language syntax.

Threading and Synchronization	Java	C#
Threads	Yes	Yes
Thread pool	Yes	Yes
Task-based parallelism	Yes[81]	Yes[82]
Semaphores	Yes	Yes
Monitors	Yes	Yes
Thread-local variables	Yes	Yes; ThreadStaticAttribute and ThreadLocal<T> class

Task-based parallelism for C#

With .NET Framework 4.0, a new task-based programming model was introduced to replace the existing event-based asynchronous model. The API is based around the `Task` and `Task<T>` classes. Tasks can be composed and chained.

By convention, every method that returns a `Task` should have its name postfixed with *Async*.

```
public static class SomeAsyncCode
{
    public static Task<XDocument> GetContentAsync()
    {
        HttpClient httpClient = new HttpClient();
        return httpClient.GetStringAsync("www.contoso.com").ContinueWith((task) => {
            string responseBodyAsText = task.Result;
            return XDocument.Parse(responseBodyAsText);
        });
    }
}

var t = SomeAsyncCode.GetContentAsync().ContinueWith((task) => {
    var xmlDocument = task.Result;
});

t.Start();
```

In C# 5 a set of language and compiler extensions was introduced to make it easier to work with the task model. These language extensions included the notion of `async` methods and the `await` statement that make the program flow appear synchronous.

```
public static class SomeAsyncCode
{
    public static async Task<XDocument> GetContentAsync()
    {
        HttpClient httpClient = new HttpClient();
        string responseBodyAsText = await httpClient.GetStringAsync("www.contoso.com");
        return XDocument.Parse(responseBodyAsText);
    }
}

var xmlDocument = await SomeAsyncCode.GetContentAsync();

// The Task will be started on call with await.
```

From this syntactic sugar the C# compiler generates a state-machine that handles the necessary continuations without developers having to think about it.

Task-based parallelism for Java

Java supports Threads since JDK 1.0. Java offers a high versatility for running threads, often called tasks. This is done by implementing a functional interface (a `java.lang.Runnable` interface) defining a single void no-args method as demonstrated in the following example:

```
Thread myThread = new Thread(() -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
});

myThread.start();
```

Also, it's possible extending `java.lang.Thread` as below:

```
class MyThread extends Thread {
    public void run() {
        String threadName = Thread.currentThread().getName();
        System.out.println("Hello " + threadName);
    }
}

MyThread myThread = new MyThread();
myThread.start();
```

Similar to C#, Java has since version 5 a higher level replacement for working with threads directly. `Executors` are capable of running asynchronous tasks and typically manage a pool of threads. All threads of the internal pool will be reused under the hood for revenant tasks, so we can run as many concurrent tasks as we want throughout the life-cycle of our application with a single executor service.

This is how the first thread-example looks like using executors:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
//executor submit a Runnable as lambda
executor.submit(() -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
});
```

In addition to `Runnable`, `Executors` supports a `Callable` interface, another functional interface like `Runnable` but returns a value.

```
public static class SomeAsyncCode {

    public static Future<String> getContentAsync(){
        ExecutorService executor = Executors.newFixedThreadPool(1);
        return executor.submit(() -> {
            return new Scanner(new URL("http://wikipedia.org").openStream(), "UTF-8").useDelimiter("\\A").next();
        });
    }

}
```

Calling the method `get()` blocks the current thread and waits until the callable completes before returning the value (in the example, a web page content):

```
//like C#'s "await" keyword
String webPageResult = SomeAsyncCode.getContentAsync().get();
```

A better practice could be ask to the users waiting while the task is completed:

```
Future<String> future = SomeAsyncCode.getContentAsync();
while(!future.isDone()){
    //ask user waiting
}

String webPageResult = future.get();
```

Additional features

Numeric applications

To adequately support applications in the field of mathematical and financial computation, several language features exist.[83]

Javas strictfp keyword enables strict floating-point calculations for a region of code. Strict floating-point calculations require that even if a platform offers higher precision during calculations, intermediate results must be converted to single/double. This ensures that strict floating point calculations return exactly the same result on all platforms. Without strict floating point a platform implementation is free to use higher precision for intermediate results during calculation. C# allows an implementation for a given hardware architecture to always use a higher precision for intermediate results if available, i.e. C# does not allow the programmer to optionally force intermediate results to use the potential lower precision of float/double.[84]

Although Java's floating point arithmetic is largely based on IEEE 754 (Standard for Binary Floating-Point Arithmetic), certain features are unsupported even when using the strictfp modifier, such as Exception Flags and Directed Roundings, abilities mandated by IEEE Standard 754 (see Criticism of Java, Floating point arithmetic).

C# provides a built-in decimal type,[85] which has higher precision (but less range) than the Java/C# double. The decimal type is a 128-bit data type suitable for financial and monetary calculations. The decimal type can represent values ranging from 1.0×10^{-28} to approximately 7.9×10^{28} with 28-29 significant digits.[86] The structure uses C# operator overloading so that decimals can be manipulated using operators such as +, -,*and /, like other primitive data types.

The `BigDecimal (https://docs.oracle.com/javase/9/docs/api/java/math/BigDecimal.html)` and `BigInteger (https://docs.oracle.com/javase/9/docs/api/java/math/BigInteger.html)` types provided with Java allow arbitrary-precision representation of decimal numbers and integer numbers, respectively. Java standard library does not have classes to deal with complex numbers.

The BigInteger,[3] and Complex[87] types provided with C# allow representation and manipulation of arbitrary-precision integers and complex numbers, respectively. The structures use C# operator overloading so that instances can be manipulated using operators such as +, -,*and /, like other primitive data types. C# standard library does not have classes to deal with arbitrary-precision floating point numbers (see software for arbitrary-precision arithmetic).

C# can help mathematical applications with the `checked` and `unchecked` operators that allow the enabling or disabling of run-time checking for arithmetic overflow for a region of code.

Language integrated query (LINQ)

C#s Language Integrated Query (LINQ) is a set of features designed to work together to allow in-language querying abilities and is a distinguishing feature between C# and Java.

LINQ consists of the following features:

- Extension methods allow existing interfaces or classes to be extended with new methods. Implementations can be shared or an interface can have a dedicated implementation.
- Lambdas allow for expression of criteria in a functional fashion.
- Expression trees allow a specific implementation to capture a lambda as an abstract syntax tree rather than an executable block. This can be utilized by implementations to represent criteria in a different language, e.g. in the form of an SQL where clause as is the case with e.g. Linq, LINQ to SQL.
- Anonymous types and type inference supports capturing and working with the result type of a query. A query may both join and project over query sources that may lead to a result type that cannot be named.
- Query expressions to support a syntax familiar to SQL users.
- Nullable (lifted) types to allow for a better match with query providers that support nullable types, like e.g. SQL.

Native interoperability

Native interoperability	Java	C#
Cross-language interoperability	Yes (with Nashorn, CORBA, JNI or JNA)[88]	Yes; C# was designed for it[88]
External/native methods	Yes	Yes
Marshalling	External glue code needed	Yes; metadata controlled
Pointers and arithmetics	Yes[89]	Yes
Native types	Yes[90]	Yes
Fixed-size buffers	No	Yes
Explicit stack allocation	No	Yes
Address-of	No	Yes
Object pinning (fix variable to address)	No	Yes

The Java Native Interface (JNI) feature allows Java programs to call non-Java code. However, JNI does require the code being called to follow several conventions and imposes restrictions on types and names used. This means that an extra adaption layer between legacy code and Java is often needed. This adaption code must be coded in a non-Java language, often C or C++. Java Native Access (JNA) allows easier calling of native code that only requires writing Java code, but comes at a performance cost.

In addition, third party libraries provide Java-Component Object Model (COM) bridging, e.g., JACOB (free), and J-Integra for COM (proprietary).

.NET Platform Invoke (P/Invoke) offers the same ability by allowing calls from C# to what Microsoft terms unmanaged code. Through metadata attributes the programmer can control exactly how the parameters and results are marshalled, thus avoiding the external glue code needed by the equivalent JNI in Java. P/Invoke allows almost complete access to procedural APIs (such as Win32 or POSIX), but limited access to C++ class libraries.

In addition, .NET Framework also provides a .NET-COM bridge, allowing access to COM components as, if they were first-class .NET objects.

C# also allows the programmer to disable the normal type-checking and other safety features of the CLR, which then enables the use of pointer variables. When using this feature, the programmer must mark the code using the `unsafe` keyword. JNI, P/Invoke, and "unsafe" code are equally risky features, exposing possible security holes and application instability. An advantage of unsafe, managed code over P/Invoke or JNI is that it allows the programmer to continue to work in the familiar C# environment to accomplish some tasks that otherwise would require calling out to unmanaged code. An assembly (program or library) using unsafe code must be compiled with a special switch and will be marked as such. This enables runtime environments to take special precautions before executing potentially harmful code.

Runtime environments

Java (the programming language) is designed to execute on the Java platform via the Java Runtime Environment (JRE). The Java platform includes the Java virtual machine (JVM) and a common set of libraries. The JRE was originally designed to support interpreted execution with final compiling as an option. Most JRE environments execute fully or at least partially compiled programs, possibly with adaptive optimization. The Java compiler produces Java bytecode. Upon execution the bytecode is loaded by the Java runtime and either interpreted directly or compiled to machine instructions and then executed.

C# is designed to execute on the Common Language Runtime (CLR). The CLR is designed to execute fully compiled code. The C# compiler produces Common Intermediate Language instructions. Upon execution the runtime loads this code and compiles to machine instructions on the target architecture.

Examples

Input/output

Example illustrating how to copy text one line at a time from one file to another, using both languages.

Java

```java
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;

public class FileIOTest {

    public static void main(String[] args) throws Exception {
        List<String> lines = Files.readAllLines(Paths.get("input.txt"));
        Files.write(Paths.get("output.txt"), lines);
    }
}
```

C#

```c#
using System.IO;

class FileIOTest
{
    public static void Main(string[] args)
    {
        var lines = File.ReadLines("input.txt");
        File.WriteAllLines("output.txt", lines);
    }
}
```

Notes on the Java implementation:

- Files.readAllBytes method returns a byte array, with the content of the text file, Files has also the method readAllLines, returns a List of Strings.
- Files.write method writes a byte array into an output file, indicated by a Path object.
- Files.write method also takes care of buffering and closing the output stream.
- An explicit option (optional last argument, using constants of StandardOpenOption) should be specified for the Files.write method whether it should overwrite or append to the output file.

Notes on the C# implementation:

- The `ReadLines` method returns an enumerable object that upon enumeration will read the file one line at a time.
- The `WriteAllLines` method takes an enumerable and retrieves a line at a time and writes it until the enumeration ends.
- The underlying reader will automatically allocate a buffer, thus there is no need to explicitly introduce a buffered stream.
- `WriteAllLines` automatically closes the output stream, also in the case of an abnormal termination.

Integration of library-defined types

C# allows library-defined types to be integrated with existing types and operators by using custom implicit/explicit conversions and operator overloading as illustrated by the following example:

Java

```java
BigInteger bigNumber =
    new BigInteger("12345678901234567890123456789O");

BigInteger answer = bigNumber.multiply(new BigInteger("42"));
BigInteger square = bigNumber.multiply(bigNumber);
BigInteger sum = bigNumber.add(bigNumber);
```

C#

```c#
var bigNumber =
    BigInteger.Parse("12345678901234567890123456789O");

var answer = bigNumber*42;
var square = bigNumber*bigNumber;
var sum = bigNumber + bigNumber;
```

C# delegates and equivalent Java constructs

Java

```java
// a target class
class Target {
    public boolean targetMethod(String arg) {
        // do something
        return true;
    }
}

// usage
void doSomething() {
    // construct a target with the target method
    Target target = new Target();

    // method reference with the Function interface
    Function<String, Boolean> ivk = target::targetMethod;

    // invoke the method
    boolean result = ivk.apply("argumentstring");
}
```

C#

```c#
// a target class
class Target
{
    public bool TargetMethod(string arg)
    {
        // do something
        return true;
    }
}

// usage
void DoSomething()
{
    // construct a target with the target method
    var target = new Target();

    // capture the delegate for later invocation
    Func<string, bool> dlg = target.TargetMethod;

    // invoke the delegate
    bool result = dlg("argumentstring");
}
```

Type lifting

Java

```java
Optional<Integer> a = Optional.of(42);
Optional<Integer> b = Optional.empty();

// orElse(0) returns 0 if the value of b is null
Integer c = a.get()*b.orElse(0);
```

C#

```c#
int? a = 42;
int? b = null;

// c will receive the null value
// because*is lifted and one of the operands are null
int? c = a*b;
```

Interoperability with dynamic languages

This example illustrates how Java and C# can be used to create and invoke an instance of class that is implemented in another programming language. The "Deepthought" class is implemented using the Ruby programming language and represents a simple calculator that will multiply two input values (`a` and `b`) when the `Calculate` method is invoked.

Java

```java
// initialize the engine

ScriptEngineManager factory = new ScriptEngineManager();
Invocable invocable = (Invocable) factory.getEngineByName("jruby");

FileReader fr = new FileReader("Deepthought.rb");
engine.eval(fr);
```

C#

```c#
// initialize the engine

var runtime = ScriptRuntime.CreateFromConfiguration();
dynamic globals = runtime.Globals;

runtime.ExecuteFile("Deepthought.rb");
```

```java
// create a new instance of "Deepthought" calculator
Object calcClass = engine.eval("Deepthought");
Object calc = invocable.invokeMethod(calcClass, "new");

// set calculator input values
invocable.invokeMethod(calc, "a=", 6);
```

```c#
// create a new instance of "Deepthought" calculator
var calc = globals.Deepthought.@new();

// set calculator input values
calc.a = 6;
calc.b = 7;
```

```
invocable.invokeMethod(calc, "b=", 7);

// calculate the result
Object answer = invocable.invokeMethod(calc, "Calculate");
```

```
// calculate the result
var answer = calc.Calculate();
```

Notes for the Java implementation:

- Ruby accessors names are generated from the attribute name with a = suffix. When assigning values, Java developers must use the Ruby accessor method name.
- Dynamic objects from a foreign language are not first-class objects in that they must be manipulated through an API.

Notes for the C# implementation:

- Objects returned from properties or methods of `dynamic` objects are themselves of `dynamic` type. When type inference (the `var` keyword) is used, the variables calc and answer are inferred dynamic/late-bound.
- Dynamic, late-bounds objects are first-class citizens that can be manipulated using C# syntax even though they have been created by an external language.
- `new` is a reserved word. The `@` prefix allows keywords to be used as identifiers.

Fibonacci sequence

This example illustrates how the Fibonacci sequence can be implemented using the two languages. The C# version takes advantage of C# generator methods. The Java version takes the advantage of `Stream` interface and method references. Both the Java and the C# examples use K&R style for code formatting of classes, methods and statements.

Java

```
// The Fibonacci sequence
Stream.generate(new Supplier<Integer>() {
    int a = 0;
    int b = 1;

    public Integer get() {
        int temp = a;
        a = b;
        b = a + temp;
        return temp;
    }
}).limit(10).forEach(System.out::println);
```

C#

```
// The Fibonacci sequence
public IEnumerable<int> Fibonacci() {
    int a = 0;
    int b = 1;

    while (true) {
        yield return a;
        yield return b;
        a += b;
        b += a;
    }
}
```

```
// print the 10 first Fibonacci numbers
foreach (var it in Fibonacci().Take(10)) {
    Console.WriteLine(it);
}
```

Notes for the Java version:

- The Java 8 Stream interface is a sequence of elements supporting sequential and parallel aggregate operations.
- generate method returns an infinite sequential unordered stream where each element is generated by the provided Supplier.
- limit method returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length.
- forEach performs an action for each element of this stream, this action could be a lambda or a method reference.

Notes for the C# version:

- The infinite Fibonacci sequence is represented by the `Fibonacci` method.
- The method is defined as returning instances of the interface `IEnumerable<int>`, which allows client code to repeatedly request the next number of a sequence.
- The `yield` keyword converts the method into a generator method.
- The method body calculates and returns Fibonacci numbers.
- The `yield return` statement returns the next number of the sequence and creates a continuation so that subsequent invocations of the `IEnumerable` interface's `MoveNext` method will continue execution from the following statement with all local variables intact.
- The implementation uses two `yield return` statements to alternate calculations instead of using a temporary tmp variable.

See also