JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002
Vol. 1, No. 5, November-December 2002

C# and Java: The Smart Distinctions

Dominik Gruntz
Northwestern Switzerland University of Applied Sciences,
Aargau, Switzerland

Part of the new .NET platform from Microsoft is the new programming language C# which was first presented in 1999. In many articles, C# was compared with Java and C++, and its new and novel features were presented and discussed. In this article we do not want to elaborate these language extensions (as e.g. properties, events, attributes, operator overloading, etc.) but rather concentrate on some subtle and almost imperceptible language changes in which C# differs from Java. These language refinements enable the compiler to mark potential problems which are otherwise only found by static analyzer tools (as e.g. lint). Programmers can no longer fall into prominent language trap doors and thus save development and debugging time. We hope, that some of these language enhancements find their way back to the Java language.

1 C# AND ITS ANCESTORS

Microsoft .NET is a platform for software components and web services. The kernel of the .NET framework is the Common Language Runtime (CLR) which can load and execute components and which provides some basic infrastructure as e.g. garbage collection, threading and security support. The components can be implemented in a variety of .NET compliant programming languages and are compiled into the Microsoft Intermediate Language (MSIL), a byte code comparable to Java byte code.

The predestinate language to program on the .NET platform is C#, a modern, object-oriented and type safe language which (according to Microsoft evangelists) combines the power and efficiency of C++ with the simplicity of Visual Basic. C# was designed by Anders Hejlsberg [Hejlsberg00], author of Object Pascal and Delphi (for an introduction to C# see e.g. [Wiltamuth00]).

Although Microsoft always declared C# as a direct successor of C++, the language has inherited almost all features of Java: garbage collection, byte code, meta information and reflection, single superclassing, interfaces, exceptions, etc. Many authors therefore compared C# with its sibling Java and discussed the new and novel features of C# [Albahari00, Eaddy01, Johnson00, Krikorian01, Kurniawan01, Surveyer00a, Surveyer00b]. Among the new features of C# are delegates as a se-

Cite this article as follows: Dominik Gruntz: *C# and Java: The Smart Distinctions*, in Journal of Object Technology, vol. 1, no. 5, pages 163–176.
http://www.jot.fm/issues/issue_2002_11/article4

cure form of function pointers (inherited from Delphi/J++), intrinsic property and event support, enum types, a restricted form of operator overloading, structs (user definable value types) and the call by reference parameter passing mode. Moreover, the type system of C# is unified (both value and reference types inherit from a common base class `object`) and value objects are automatically converted to reference objects (boxing and unboxing) when necessary, e.g. it is possible to store values of a primitive type into a collection which collects objects of type `object`.

In this paper we do not want to repeat the presentation and discussion of the new language features of C#, but rather want to concentrate on simple and small distinctions to Java that fix common programming trap doors. Some of them are so subtle that they are only recognized when the language report is carefully read. Cautious programmers will not become aware of these language changes but they enable the compiler to mark potential problems and to force the programmer to clarify its statements. This way, the compiler takes over some of the tasks of static analyzer tools as e.g. lint for the C/C++ programming language. One example, which was discussed in some comparison articles, is the change of the switch statement that fixes the well-known fall-through problem of C++ and Java. There are other such fine changes in the language that we are going to present now, and hopefully, we will see some of these features in a future version of Java.

2 SWITCH STATEMENT

The `switch` statement is used to conditionally execute statement blocks. The value of an expression defines which `case` is executed. Both Java and C++ allow the control flow to implicitly fall through different cases, i.e., if a `case` is not explicitly left, the `switch` statement keeps executing through all `case`s up to the end of the statement. The following statement is legal in Java and C++:

```
switch(direction){
   case 0: str = "North";
   case 1: str = "East";
   case 2: str = "South";
   case 3: str = "West";
}
```

The intention of the programmer was to convert the value of `direction` into the corresponding cardinal point, but instead the variable `str` always gets assigned the string `"West"` (provided that the value of `direction` is either 0, 1, 2 or 3). The problem is that a `break` statement has to be added after each `case` block.

C# fixes this problem with a small change in the language definition: it requires that each `case` block in a `switch` statement has to be left explicitly, e.g. with a `break`, `return`, `continue` or a `goto` statement. Otherwise the compiler issues the error message “Control cannot fall through from one case label to another”.

However, C# still supports the old mode, i.e., if a programmer really wants to have different entry points he can explicitly jump to another `case` label (note that multiple labels on the same `case`-section are still possible):

```
switch(s1.CompareTo(s2)) {
   case -1: // s1 < s2
      swap(ref s1, ref s2);
      goto case 1;
   case 0: // s1 = s2 or
   case 1: // s1 > s2
      process(s1, s2);
      break;
}
```

3 TRY-FINALLY

The next language change concerns the `try-finally` statement. This construct is e.g. used to guarantee that a database connection is closed or that a monitor is properly exited. Independent of how the `try` block is left, the statements in the `finally` block are always executed. The `try` block is left when one of the following situations takes place:

1. The `try` block finishes normally.
2. An exception is thrown in the `try` block.
3. A `return`, `break` or `continue` statement is executed in the `try` block.

The latter two situations are problematic in case that an additional `return` statement is executed in the `finally` block. Consider the following Java example:

```
public class TryFinallyDemo {

   public static int method1 () {
      try {
         return 3;
      }
      finally {
         return 4;
      }
   }
```

```
    public static int method2 () {
        try {
            throw new UnsupportedOperationException();
        }
        finally {
            return 2;
        }
    }

    public static void main(String[] args) {
        System.out.println(method1());   // Output: 4
        System.out.println(method2());   // Output: 2, no exception
    }
}
```

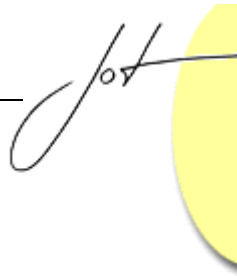
The output that is generated is 4 for the call of `method1` and 2 for the call of `method2`; an exception is not thrown. The statement of the `try` block in `method1` is about to return the result 3 and in `method2` about to throw a runtime exception, but before these statements are executed the `finally` block is executed which contains a `return` statement in both cases. This `return` statement overshadows the regular leaving of the `try` block, i.e., the `return` or the `throw` statement, respectively.

Traditionally, programmers expect that a method is left when a `return` statement is executed — this is no longer true with the usage of `finally` in Java. An experienced Java programmer may argue that the behavior is obvious as when reading a `try` statement in Java the corresponding `finally` block has to be considered as well. The above code can be interpreted as if the statements in the `finally` block are executed before the `return` or `throw` statement is executed in the `try` block. With this interpretation in mind the two methods `method1` and `method2` can be read as follows and the behavior of the program becomes obvious. One could argue, that the Java compiler should issue an "unreachable code detected" error.

```
public static int method1 () {
    return 4;
    return 3;
}

public static int method2 () {
    return 3;
    throw new UnsupportedOperationException();
}
```

This interpretation is correct for the above example, but which output is generated in the following example?



```
public class TryFinallyDemo {

   private static int counter = 0;

   public static int getCounter () {
      try {
         return ++counter;
      }
      finally {
         return ++counter;
      }
   }

   public static void main(String[] args) {
      System.out.println(getCounter());
   }
}
```

The private field `counter` is initialized with zero, and for both `return` statements, the result is computed and placed on the stack. Thus, the counter is incremented twice and the result which is returned from the `getCounter` call is 2.

Haggar described this problem in his book in Praxis 22 [Haggar00]. He wrote that this particular characteristic of `finally` in Java has the potential to cause confusion and may lead to long debugging sessions. To avoid this pitfall, he recommends to not issue a `return`, `break` or `continue` statement inside a `try` block or otherwise to secure, that the existence of a `finally` block does not change the return value of a method.

In C# this problem is fixed as the language does not allow to leave the control flow of a `finally` block with a `return`, `break`, `continue` or `goto` statement. When the above examples are compiled, the C# compiler issues an error as shown in the following listing.

```
public class TryFinallyDemo {
   public static int method1 () {
      try {
         return 3;
      }
      finally {
         return 4;
         // error CS0157:
         // Control cannot leave the body of a finally clause
      }
   }
```

```
    public static void Main() {
        System.Console.WriteLine(method1());
    }
}
```

4 MEMBER DECLARATIONS

The next pitfall is one we stumbled in ourselves and it took us about one hour debugging time to find the problem (ok, probably it was also late in the evening). We demonstrate this problem in the context of a simple phone book implementation. The phone book stores phone numbers which can be queried with `lookupNumber` and new entries can be entered with `insertNumber`. The entries are stored in a hash map. The phone book is initialized in its constructor with data from a data base. To simplify the code two entries (phone numbers of Daffy Duck and Huey Lewis) are entered when a new instance of `PhoneBook` is created.

```
public class PhoneBook {
    public void PhoneBook() {
        // load data from a data basis
        insertNumber("Daffy Duck", "310-555-1212");
        insertNumber("Huey Lewis", "415-555-1212");
    }

    private java.util.Map entries = new java.util.HashMap();

    public void insertNumber(String name, String number) {
        entries.put(name, number);
    }

    public String lookupNumber(String name) {
        return (String)entries.get(name);
    }

    public static void main(String[] args) {
        PhoneBook book = new PhoneBook();
        String name = args[0];
        String number = book.lookupNumber(name);
        if(number == null)
            System.out.println(name + " is not stored in the phone book");
        else
            System.out.println(name + " has number " + number);
    }
}
```

When the program is executed it always reports that the given entry is not stored in the phone book, independent of the supplied query argument.

```
Java> java PhoneBook "Daffy Duck"
Daffy Duck is not stored in the phone book
```

Probably, you have already found the source of this problem, but the program we had to debug was really much more complicated and we looked at everything except at the most obvious. The problem is, that the constructor we defined for the class `PhoneBook` is not a constructor, it is a regular method which simply has the same name as the class in which it is defined. A constructor does not define a result type. Unfortunately, the pretended "constructor" had no arguments and no other constructors were defined, so Java silently used the default constructor.

What a silly idea to name a method with the same name as the class in which it is defined. Only paranoid programmers will do that. Agreed! But C# goes a step beyond and forbids to define methods which such a name! With C#, the compiler would have marked our program with the error "member names cannot be the same as their enclosing type" and would have informed us, that what we intended to define as constructor is in fact not a constructor.

```
class PhoneBook {
   public void PhoneBook() {
   }
}
```

```
C#> csc PhoneBook.cs
C# compiler error:
PhoneBook.cs(2,14): error CS0542: 'PhoneBook': member names cannot be
        the same as their enclosing type
PhoneBook.cs(1,7): (Location of symbol related to previous error)
```

5 CONSTRUCTORS

In the context of constructors we found another problem in Peter Haggar's "Practical Java" book [Haggar00]. In Praxis 68 he recommends: Use care when calling non-final methods from constructors. The problem is, that such a non-final method may be overwritten in an extending class and thus code may be executed on not yet initialized parts of a class (where not yet initialized in Java means system initialized with zeros). Consider the following example (adapted from [Haggar00]) where the constructor of the base class `DB` calls the `lookup` method to retrieve some data from

a database (to simplify the code the `lookup` method returns 5). In a derived class the `lookup` method is overwritten and returns the value of the field `num`.

```
class DB {
   private int val = lookup ( );
   public  int lookup ( ) {
      /* in reality perform DB lookup here */
      return 5;
   }
   public final int value ( ) { return val; }
}

class DerivedDB extends DB {
   private int num = 10;
   public  int lookup ( ) { return num; }
}

class Test {
   public static void main ( String[] args) {
      DB db1 = new DB ( );
      DB db2 = new DerivedDB ( );
      System.out.println("db1.value() returns "+ db1.value ( ) );
      System.out.println("db2.value() returns "+ db2.value ( ) );
   }
}
```

When executed, the program prints the following output:

```
Java> java Test
db1.value() returns 5
db2.value() returns 0
```

The call of `db1.value()` returns 5 as expected, but the expected result for the query of `db2`'s value is 10 (the initial value returned by `DerivedDB`'s `lookup`). The problem with the initialization of instances of `DerivedDB` is, that the `lookup` method is called from an initializer which is called from the constructor `DB()` which itself is called from the default constructor of the class `DerivedDB` *before* the field `num` is initialized with the value 10. The `lookup` method in `DerivedDB` therefore returns the system default value of `num` which is zero, and this is the value which is stored in the `val` field.

The point is, that when a field declaration is read which contains a variable initializer as in

```
private int num = 10;
```

the reader assumes that this variable is initialized right from the beginning with the value 10. In Java, this expectation is not guaranteed as we just have seen.

The above Java code works as expected if the variable num is declared as final (then the value is resolved by the compiler and the field doesn't exist at runtime) or if it is declared as static (then the field is initialized when the class is loaded).

If the above example were translated to C++, it would print the result 5 for both cases. The reason is, that in C++ virtual calls from a constructor do only call methods up to the already initialized level — for C++ this is very momentous as the fields do not have guaranteed default initial values as in Java and C#, i.e., pointers would refer to somewhere in the pampas.

In C#, the above example returns the expected results.

```
class DB {
   private int val;
   public DB ( ) { val = lookup ( ); }
   public virtual int lookup ( ) {
      /* perform DB lookup */
      return 5;
   }
   public int value ( ) { return val; }
}

class DerivedDB : DB {
   private int num = 10;
   public override int lookup ( ) { return num; }
}

class Test {
   public static void Main ( ) {
      DB db1 = new DB ( );
      DB db2 = new DerivedDB ( );
      System.Console.WriteLine("db1.value(): 0", db1.value());
      System.Console.WriteLine("db2.value(): 0", db2.value());
   }
}
```

```
C#> Test
db1.value() returns 5
db2.value() returns 10
```

The reason is, that in C# the constructors are generated differently as in Java. First, all instance field initializers are executed, second the constructor of the base class is called and third the body of the constructor itself is executed. In Java, the

base class constructor is called before the initializers of the fields are executed. So C# achieves with a simple language change that the code meets the programmers expectations. Again, the Java behavior can be simulated in C# by initializing the field num in the body of the constructor.

```
class DerivedDB : DB {
   private int num;
   public DerivedDB () {
      num = 10;
   }
   public override int lookup ( ) { return num; }
}
```

You may ask why Java is defined the way it is defined. Why are the field initializers not executed before the constructor of the base class is called as in C#? The reason is, that the initializers may contain arbitrary method calls; in particular you could refer to methods or fields in the base class which would then not be initialized. Back to square one! Has C# a serious defect here? The answer is no, as in C# the expressions for field initializers are restricted. In C#, an instance field initializer cannot reference the instance being created. It is an error to reference methods, fields or properties of the same instance from within an instance field initializer. This restriction applies both to members of the current class and of the base class. This rule avoids that instance field initializer access uninitialized fields, either directly or over method calls. The direct initialization of the field num with a call to lookup (which is allowed in Java) would not be possible in C#.

```
class DB {
   private int val = lookup();
   public virtual int lookup ( ) { return 5; }
   public int value ( ) { return val; }
}
```

```
C#> csc DB.cs
DB.cs(2,20): error CS0236: A field initializer cannot reference the
        nonstatic field, method, or property 'DB.lookup()'
```

We think, that the solution C# has found is a compromise which fits most programmer's expectations.

6 STATIC MEMBERS

Another restriction in the C# language compared to Java is the access to static class members. In Java, static methods or fields can either be accessed with the

class name or over an instance. In most books the latter form is not recommended as it may confuse the reader of a program as in the following example:

```
class B {
   public static void foo() { System.out.println("B::foo"); }
   public void bar() { System.out.println("B::bar"); }
}

class D extends B {
   public static void foo() { System.out.println("D::foo"); }
   public void bar() { System.out.println("D::bar"); }
}

class Test {
   public static void main (String[] args ) {
      B b = new D();
      b.foo();      // Output: B::foo
      b.bar();      // Output: D::bar
      b = null;
      b.foo();      // Output: B::foo
   }
}
```

It may be confusing to the reader of this program that the first method call (`b.foo`) is bound statically, whereas the second call (`b.bar`) is bound dynamically. Additionally, the specification of an object in the call `b.foo()` is also confusing as the object is not passed; it may even be a `null`; only its (static) type is used by the compiler to determine which static method to invoke. More confusing calls are listed in the following class.

```
class Test {
   public static void main (String[] args ) {
      B b = null;
      b.foo();            // Output: B::foo
      ((D)null).foo();    // Output: D::foo
   }
}
```

Static methods can also be called with the class name directly (`B.foo()` and `D.foo()` in our example) and most Java text books recommend this explicit notation. C# translates this recommendation into the language definition. Static members cannot be accessed over an instance. An explicit type qualification has to be specified whenever a static member is accessed from outside the class. Inside a method a static member of the same class can still be accessed without class prefix.

7 INNER CLASSES

Both Java and C# support the declaration of inner classes, i.e., the declaration of a class within the name scope of another class. (In Java these classes are called static inner classes. Element classes which contain an implicit reference to an instance of the outer class are not supported in C#.) However, Java makes a distinction between the namespace of fields, methods and nested classes within a class, i.e., it is possible to declare a field, a method and a static inner class with the same name. As nested classes and fields are accessed with the same scope resolution operator, access may be difficult if both a field and a nested class are declared with the same name.

```
class X {
   static class Y {
      static String Z = "Romeo";
   }

   static C Y = new C();
}

class C {
   String Z = "Julia";
}

public class JavaNestedClassesConsideredHarmful {
   public static void main(String[] args) {
      System.out.println(X.Y.Z);
   }
}
```

Surprisingly, this program can be translated without problems and upon execution the string `"Julia"` is printed (the question how `"Romeo"` can be accessed from the main program without changing the declarations of classes `X` and `C` is left as an exercise to the reader — access is possible).

In C# this problem is also fixed. All declarations in a class fall into the same namespace, thus it is not possible to declare a field and a nested class with the same name. C# also prohibits the declaration of fields and methods with the same name. This restriction is necessary as methods are accessed like fields in a delegate instanciation.

8 SUMMARY

In this article we have presented some subtle changes in the definition of C# compared to Java. Although these changes theoretically confine the programmer (no

switch fall-through, no methods with the same name as the class, restricted initializers, etc.), we claim that these restrictions do not hurt normal programmers but rather help them to avoid custom errors.

We also have the impression that the designers of C# suffered themselves from the C++/Java pitfalls described in this article — otherwise they would never had the idea and motivation to fix them. Engineering a programming language really required in-depth programming experience.

If these features were included in the Java language, we claim that 98% of all programs could still be compiled, but some potential problems and pitfalls would already be marked by the compiler. This would be a further step in the direction to a language which enables programs with less errors. Java has already made big progress compared to C++ as e.g. uninitialized variables are marked or as the free manipulation of pointers is no longer supported. Hopefully, Sun will "steal back" some of C#'s innovations described in this article.

REFERENCES

[Haggar00] Peter Haggar: Practical Java Programming Language Guide, Addison Wesley, 2000, ISBN 0-201-61646-7.

[Albahari00] Ben Albahari: A Comparative Overview of C#, Genamics, Released 31 July, 2000, http://genamics.com/developer/csharp_comparative.htm.

[Eaddy01] Marc Eaddy: C# Versus Java, Dr. Dobb's Journal, February 2001, p. 74, http://www.ddj.com/documents/s=870/ddj-0102g/0102g.htm.

[Hejlsberg00] Anders Hejlsberg & Scott Wiltamuth, C# Language Reference, Microsoft Corp, 2000, http://msdn.microsoft.com/library/en-us/csspec/html/CSharpSpecStart.asp.

[Johnson00] Mark Johnson: C#: A language alternative or just J--?, Java World, November/December 2000, http://www.javaworld.com/jw-11-2000/jw-1122-csharp1.html.

[Krikorian01] Raffi Krikorian: Contrasting C# and Java Syntax, Published on The O'Reilly Network, 2001, http://www.oreillynet.com/pub/a/dotnet/2001/06/14/csharp_4_java.html.

[Kurniawan01] Budi Kurniawan: Comparing C# and Java, Published on The O'Reilly Network, 2001, http://www.oreillynet.com/pub/a/dotnet/2001/06/07/csharp_java.html.

[Surveyer00a] Jacques Surveyer, C# Strikes a Chord, Dr. Dobb's Journal, September 5, 2000, http://www.ddj.com/documents/s=875/ddj0065g/.

[Surveyer00b] Jacques Surveyer, C# = (C-Sharp == Microsoft Java++)? True : False; Java Report, October 2000.

[Wiltamuth00] Scott Wiltamuth, The C# Programming Language, Dr. Dobb's Journal, October 2000, p. 21.

ABOUT THE AUTHOR

Dominik Gruntz is professor at the Northwestern Switzerland University of Applied Sciences. His focus is on component software and framework design. He is a member of the enterprise computing research group and leads the mobile enterprise project. He can be reached at d.gruntz@fh-aargau.ch and on the Web at http://www.cs.fh-aargau.ch/~gruntz.