Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

C#: introduction to object-oriented features

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

C# classes and objects

Classes and objects

- The basic encapsulation unit is the class
  - as in every object-oriented language
- A class is made of a number of features (or members)
  - fields (instance variables)
  - methods
- Classes and features have different levels of visibility
- Objects are class instances
  - and classes are sets of objects
    - or blueprints for creating objects
  - constructors are special methods to create new objects
  - in C#, objects are automatically destroyed when no longer referenced (garbage collection)
    - destructor syntax exists, but to create finalizer methods

A simple class example

```
namespace JavaCsharpInDepth
{
      using System;

      public class MainClass {
            /// <author> John H. Doe </author>
                  // 'Main' must be capitalized!
            public static void Main(String[] args)
            {
              Game myGame = new Game();
              Console.WriteLine("Game starts!");
              myGame.startGame();
            }
}
```

Main method

In C#, the **`Main`** **`static`** method can be:

- with argument **`String[] args`**
- without arguments

- returning **`void`**
- returning **`int`**

This is different than in Java, where the format of **`main`** is fixed

Fields (instance variables)

- Relate to a class instance
- Declared within the class curly brackets, outside any method
- Visible at least within the class scope, within any method of the class
- Automatically initialized to the default values
  - `0` or `0.0` for numeric types, `'\0'` for chars, **`null`** for references, **`false`** for booleans, the value associated with 0 for **`enum`**, a default initialization of members for **`struct`**
- Warning: in standard C# parlance, “attributes” denote a kind of annotation, not fields

Methods (instance methods)

- Relate to an instance and are declared within the class curly brackets

- May have arguments
- Must have return type (possibly **`void`**)

- Constructors are “special” (more on this later)

- Also special members in C#:
  - properties, **`delegate`**, **`event`**
  - They don’t exist in Java as such
  - More on these later

Information hiding (a.k.a. access modifiers)

Field and method visibility

- `public`: visible everywhere
- `protected`: visible within the class and in subclasses
- `internal`: visible in the same assembly (basically, the same compiled CIL file)
  - this is the default visibility for top-level types
- `internal protected`: class, subclasses, and in the same assembly
- `private`: visible only within the class
  - this is the default visibility for class members

Class visibility

- Classes can use all access modifiers except protected.

The `static` modifier

When applied to fields and methods

- Relates to a specific class, not to a class instance
- Shared by every object of a certain class
- Accessed without creating any class object

When applied to a class

- The class must contain only static fields and methods
- The class cannot be instantiated

Constructors

- Same name as the class
- No return type (not even `void`)
- An argumentless constructor is provided by default if no other constructor is explicitly given

Local variables

- Declared within a method's scope (denoted by curly brackets)
- Visible only within the method's scope
- De-allocated at method end
- Not automatically initialized
- Must be initialized before usage
  - compiler checks this in a conservative way

The keyword **this**

Refers to the current object

```
public class Card {

    private int value;

    // this is a property
    public int Value {
        get { return value; }
        set { this.value = value; }
    }
}
```

Nested classes

It's a class defined inside another class

Less expressive than Java's nested inner classes: in C#, the nesting controls visibility only, not behavior. Hence:

- There need not be a relation between instances of the nested class and instances of the containing class
- In general, the nested class cannot access members of the containing class
- A nested class can't be anonymous

C#'s delegates replace one of the main usages of Java's (anonymous) inner classes: wrappers of operations handling events

Nested classes: example usages

Nested classes may be used to:

- Declare helper classes used by the containing class but whose details are irrelevant to clients of the containing class.

```
class PersonList : IEnumerable<Person> {
        // implementation of the list
  private class PersonEnumerator :
                              IEnumerator<Person> {
    // enumerator customized for Persons
  }  // clients only know about the interface
  public IEnumerator<Person> GetEnumerator() {
        return new PersonEnumerator(this);
  }
```

- Group together a number of tightly related variants of the containing class and dispatch them to clients with static methods (as in the factory design pattern).

Method overloading

- Using the same name with different argument list
  - list can differ in length, argument type, or both
- Example: constructors
- Method signature: name + arguments list
  - The return type is not part of the signature
- Tip: overloading may reduce readability: don’t abuse it

Method overloading with subtypes

When a method name is overloaded with argument types that are related by inheritance, method resolution selects the "closest" available type.

Example: `Student` is a subtype of `Person`

```
class X {
      // v1
    void foo (Person p) { }
      // v2
    void foo (Student p) { }
}
X x = new X();
x.foo(new Person()); // Executes v1
x.foo(new Student()); // Executes v2
```

Method overloading with subtypes

When a method name is overloaded with argument types that are related by inheritance, method resolution selects the "closest" available type.

Example: `Student` is a subtype of `Person`

```
 class Y { void foo (Person p) { ... } }
 class Z { void foo (Student p) { ... } }

Y y = new Y();
y.foo(new Person()); // OK
y.foo(new Student()); // OK

Z z = new Z();
z.foo(new Person()); // Error
z.foo(new Student()); // OK
```

Operator overloading

- Operator overloading is possible with the **`operator`** keyword

```
public class Complex {

    private int re, im;

    public Complex(int re, int im) {
        this.re = re;
        this.im = im;
    }

    public static Complex operator +(Complex c1, Complex c2) {
        return new Complex(c1.re + c2.re, c1.im + c2.im);
    }
}
```

Operator overloading (cont'd)

The following operators can be overloaded:

- Unary: `+ - ! ~ ++ -- true false`
- Binary: `+ - * / % & | ^ << >> == != > < >= <=`

- If you overload a binary operator `+`, the `+=` operator is implicitly overloaded, too
  - Same for `-` and `-=`, `*` and `*=`, etc.
- Cast operators are also overloaded by defining explicit conversion operations
- At least one argument of the overloaded operator must belong to the class where the overloading definition occurs
- Operators don't have to be static and can have side effects
  - but think twice before relying on this feature!

Conversion operators

Using the keywords **explicit** and **implicit**, we can define conversion operators

```
public class Point {
    private double x, y;
    public Point(double x, double y) { ... }

    // explicit conversion: x --> (x, x)
    public static explicit operator Point(double x) {
        return new Point(x, x);
    }

    // implicit conversion: any string --> (0, 0)
    public static implicit operator Point(string s) {
        return new Point(0.0, 0.0);
    }
}
```

Conversion operators

Using the keywords **explicit** and **implicit**, we can define conversion operators

```
public class Point {
    // explicit conversion: x --> (x, x)
    public static explicit operator Point(double x) {...}
    // implicit conversion: any string --> (0, 0)
    public static implicit operator Point(string s) {...}
}

// Example client
Point p1 = (Point) 42.0;        // p1 is (42.0, 42.0)
Point p2 = "abcde";             // p2 is (0.0, 0.0)
```

Method argument passing

C# supports two argument passing semantics

- by value (the default)
- by reference (with the **`ref`** keyword)
- the "output parameter semantics" (with the **`out`** keyword) is a variant of the reference semantics

By-value argument passing

This is the default (no keywords)

- All the primitive types are passed by value
  - Inside the method body we work with a local copy
  - We return information using the **`return`** keyword
- (Object) Reference types are passed by value too, but:
  - What is passed by value is the reference (i.e, an object address)
  - Consequently, a method can change the state of the object attached to the actual arguments through the reference

By-value argument passing

This is the default (no keywords)

```
public void no_swap(int i, int j) {
    int tmp = i;
    i = j;       j = tmp;
}
...
    int a, b;
    a = 3 ; b = 5;
    no_swap(a, b);
    // a == 3  &&  b == 5
```

By-reference argument passing

With the **`ref`** keyword

- The method can modify directly the value of the actual argument in the caller
  - The caller must use the **`ref`** keyword too (rationale: it enhances the clarity of what’s going on)
- If a reference type is passed by reference the method can change the value of the reference itself in the caller

By-reference argument passing

With the `ref` keyword

```
public void swap(ref int i, ref int j) {
    int tmp = i;
    i = j;      j = tmp;
}
...
    int a, b;
    a = 3 ; b = 5;
    swap(ref a, ref b);
    // a == 5  &&  b == 3
```

Output arguments

With the **`out`** keyword

- This is meant to mark arguments used as "additional returned values"

- In practice, it achieves a semantics which is very similar to the **`ref`** keyword
- The differences:
  - a **`ref`** argument must be initialized by the caller before calling the method
  - an **`out`** argument must be written by the callee before the method returns

Variable number of arguments

To pass a variable number of arguments to a method:

- Use a collection (including arrays)
- Use a **`params`** argument

```
public void write(params String[] someStrings) {
        foreach (String aString in someStrings) {
            Console.Write(aString);
        }
}
```

- This is just syntactic sugar for an array
  - You can pass an array as actual
- The **`params`** argument must be the only one of its kind and the last one in the signature

Static constructors

- Similar to “static block initializers” in Java
- No arguments, no return type, no visibility specifiers
- The code within them is executed before the first instance of the class is created or any static member is referenced

Destructors (finalizer methods)

Any class **`C`** may include a destructor method:

`~C()`

which is syntactic sugar for overriding **`Object.Finalize`** in any class.

The destructor method is called just sometime after an object becomes inaccessible

- The destructor may not be called at all (e.g. if running process terminates first)
- No guarantee on when a destructor is called during garbage collection

What's for: do some final clean-up upon object disposal

- E.g.: resources not properly released beforehand

It is not meant for general release of resources

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Inheritance, polymorphism, and dynamic dispatching

Inheritance

- We can explicitly inherit from one class only
  - A class C inheriting from D:

    ```
    public class C : D
    ```
  - Otherwise, every class implicitly inherits from `Object`
- Visible (i.e., `public` and `protected`) inherited fields and methods are available in the heir

Overriding and dynamic dispatching

Overriding: method redefinition in a subclass

Overriding rule:

- overriding method must have the same signature and return type as in the superclass
- covariant return types are not allowed in C#

(Something similar can be obtained with genericity)

Unlike Java: static dispatching applies by default.

Overriding and dynamic dispatching

There are two types of method redefinition in C#:

- With the **`new`** keyword (hiding/shadowing)
  - dynamic dispatching does not apply
  - if you don't write **`new`** you get a warning but hiding semantic is assumed
  - can change the visibility of the method
- With the **`override`** keyword (overriding)
  - dynamic dispatching does apply
  - only allowed if method is declared as **`virtual`** in parent class
  - cannot change the visibility of the method
- An **`override`** method implicitly remains virtual until it is declared as **`sealed`**

Overriding and dynamic dispatching

```
public class A { public virtual void Do() { } }
// virtual; hence both types of redefinition
   are possible

public class B : A { public new void Do() { } }
// non-polymorphic redefinition

public class C : A { public override void Do() { } }
// polymorphic redefinition

A x = new B();  x.Do();    // static dispatching
B y = new B(); y.Do();     // static dispatching
A z = new C(); z.Do();     // dynamic dispatching
```

Casting and Polymorphism

Casting is C++/Java/C# jargon to denote polymorphic assignments.

- Let S be an ancestor of T (that is, T $\rightarrow^*$ S)
  - Upcasting: an object of type T is attached to a reference of type S
  - Downcasting: an object of type S is attached to a reference of type T

```
class Vehicle;
class Car extends Vehicle;
Vehicle v =(Vehicle)new Car(); // upcasting
Car c = (Car)new Vehicle(); // downcasting
```

Casting in C#

- Upcasting is implicit
  - For primitive types, upcasting means assigning a "smaller" type to a "larger" compatible type
    - `byte` to `short` to `int` to `long` to `float` to `double` (`long` to `float` may actually lose precision)
    - `char` to `int`
  - For reference types, upcasting means assigning a subtype to a supertype, that is:
    - a subclass to superclass
    - an implementation of an interface X to that interface X
    - an interface X to the implementation of an ancestor of X
- Downcasting must be explicit
  - can raise runtime exceptions if it turns out to be impossible
- We can use conversions (see before) to mock casts of reference types outside the inheritance hierarchy.

The `is` and `as` keywords

- The `is` keyword performs runtime checking of the dynamic type of a reference variable
  - Syntax: `aVariable is aType`
  - Is the object attached to `aVariable` compatible with `aType`?
    - Compatible means of `aType` or one of its subtypes

- The `as` keywords performs a conversion; if the conversion fails, the reference takes value `null`
  - Syntax: `aVariable as aType`
  - If `aVariable is aType` is the case, it is equivalent to:
    `(aType) aVariable`
  - Otherwise, it is equivalent to `null`

Shadowing

Variables with the same name and different (but overlapping) scopes:

- A local variable shadows a field with the same name: use `this` to access the field
- For fields, only shadowing redefinitions are allowed
  - use the `new` keyword to avoid warnings
- For methods, we've seen the two different types of redefinition

The `sealed` class modifier

- `sealed` class
  - Cannot be inherited from
- `sealed` method or field
  - Can't have further `override` (but must itself be an `override`)
  - Further `new` redefinitions are still allowed
- To have constant (local) variables: use keyword `const`

Using `new` after `sealed`

Using `new` after `sealed` is allowed, but it is as if dynamic dispatching “stops” at the sealed class:

```
class C { virtual void foo() {} }
class D : C { sealed override void foo() {} }
class E : D { new void foo() {} }

E v1 = E();
C v2 = new D();
v1.foo();  // calls definition in E
v2.foo();  // calls definition in D
C v3 = new E();
v3.foo();  // calls definition in D
```

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

The object creation process

The keyword `base`

- Enables invocation of a superclass method or constructor from within an overriding method in a subclass
  - regardless of whether the overriding was with dynamic or static dispatching

- Can be used to explicitly invoke a constructor of the superclass (see next example)

Chained constructors

Any constructor implicitly starts by executing the argumentless constructor of the parent class, unless:

- A specific constructor of the superclass is invoked using `base(...)`
- Another specific constructor of the same class is invoked using `this(...)`
- `base(...)` or `this(...)` must occur after the signature of the constructor, separated by a colon

Chained constructors

```
public class CreatureCard : Card {
    int value;
    public CreatureCard(String name)
      : base(name) {
           //specific initializations
           value = 7;
      }
      public CreatureCard(int value)
      : this("Big Monster") {
           //specific initializations
           this.value = value;
      }
}
```

Object creation process

```
MyClass obj = new MyClass();
```

(`static` members are initialized before)

- `new` allocates memory for a `MyClass` instance (all attributes, including inherited ones)
- initializes all attributes to default values

If constructor references `base` (explicitly or by default):

1. Execute `MyClass`'s initializers in their textual order
2. Recursive call to constructor of superclass
3. Execute constructor body

If constructor references `this` (another constructor X):

1. Recursive call to other constructor X
2. Execute rest of originally called constructor body

Object creation process: example

```
public class Person {
      protected int age = 1;
}

public class Student : Person {
   protected double gpa;
   public Student() {
      age = 6;
      gpa = age/2 + 1.0;
   }
}

Person p1 = new Person();        // age = 1
Person p2 = new Student();       // age = 6, gpa = 4.0
```

Example (closer to intentions in Java)

```
public class Person {
      protected int age;
      public Person() : this(1) {}
      public Person(int age) { this.age = age; }
}

public class Student : Person {
  protected double gpa;
  public Student() : base(6) {
      gpa = age/2 + 1.0;
  }
}

Person p2 = new Student();        // age = 6, gpa = 4.0
```