Computer Science
Engineering School

Software
Engineering

Programming Technologies and Paradigms

The Object Oriented Paradigm

Francisco Ortín Soler

University of Oviedo

Course Material

- These slides constitute a **summary** of the theory classes in *The Object Oriented Paradigm* unit
- The C# programming language will be <u>used</u>

  <u>But it is not explained!</u>
- Students must obtain the competences related object-oriented programming in C#:
  - Considering their knowledge of Java (the *Programming Methodology* subject)
  - Doing the **mandatory autonomous activities**
  - Working in laboratory classes
  - Doing their homework

Course Material

- **Mandatory activity** to be done before the first laboratory class: Read the following slides

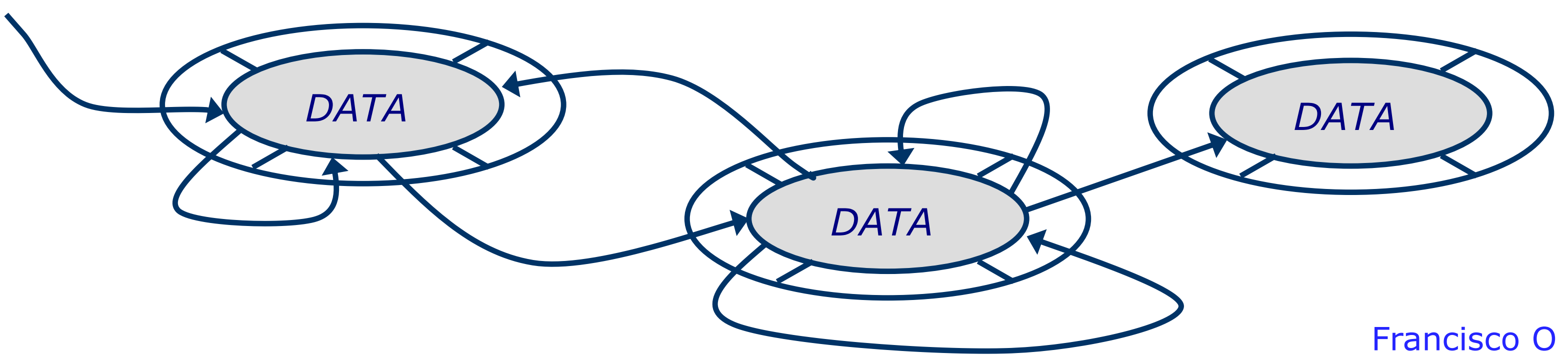
  Elements of the Object Oriented Paradigm (Activities)
- Slides refer to source code that must be **opened**, **analyzed**, **modified**, **executed** by the students, making sure that they **understand** the code
  - The source code is written in underlined blue font (e.g., basic/console)

Content

- The Object Oriented Paradigm
- Encapsulation
- Modularity
- Overloading
- Inheritance and Polymorphism
- Abstract Classes and Interfaces
- Exceptions
- Assertions
- Generics
- Type Inference

Object Oriented Paradigm

- The **object** is the main abstraction, defining **programs** as <u>interactions among objects</u>
  - An object comprises data (fields) and services (methods)
- It is based on the idea of modeling **real objects** by means of **software objects**
  - The idea is to bring the <u>domain model</u> closer to the <u>program model</u>
- An object-oriented program is made up of a set of objects that interchange messages among them



Abstraction

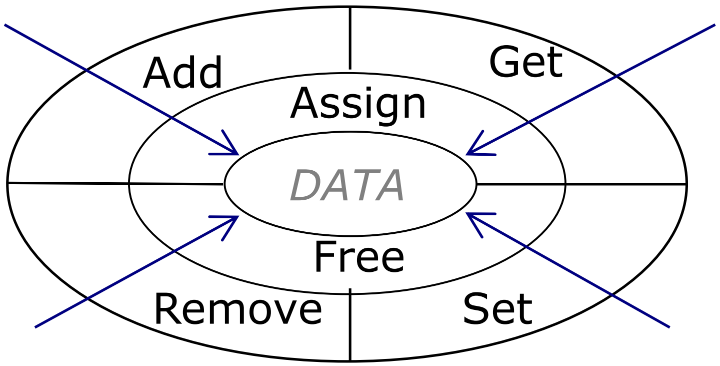
- **Abstraction**: denotes the essential characteristics of an object that distinguish it from all other kinds of objects [Booch, 1996]
  - The main mechanism of most programming languages to represent their abstractions are **types**

Encapsulation

- **Encapsulation**: Encapsulation is the process of <u>compartmentalizing</u> the elements of an abstraction that constitute its <u>structure</u> and <u>behavior</u> [Booch, 1996]
- **Information hiding** separates those elements in an abstraction accessible to the rest the application from those intern to the abstraction [Meyer, 1999]
  - Some authors include information hiding in the concept of encapsulation
- To support information hiding, programming languages provide different **levels** for **controlling access** to members (methods and fields)
- Each object is isolated from the outside, and exposes an **interface** to other objects that specifies how other objects may interact with it

Benefits of Encapsulation

- Let's suppose we are implementing a `Collection` class to collect integers, with the following interface
  - **`Add`**
  - **`Get`**
  - **`Set`**
  - **`Remove`**



- Let's suppose that our first implementation uses a linked list, and we detect its low performance for random access (i.e., `Get` and `Set`)
- How can we increase its runtime performance?

Benefits of Encapsulation (II)

- We can change the class implementation
  - A vector could provide better performance for random access
- We do not modify its interface, only its implementation
  - The rest of the application does not need to be changed!
  - Encapsulation ⇒ **Maintainability**
- Encapsulation provides a clear interface to be used in different scenarios ⇒ **Reutilization**
- A limited set of methods accesses the object data, avoiding inconsistency errors ⇒ **Robustness**

Properties

- C# offers **properties** as a flexible mechanism to access the abstract state of objects, obtaining the benefits of encapsulation
  - The object state is hidden, defining a **public interface** to its access
  - Its implementation can be changed without changing its use (**maintainability**)
- Properties can **read** and/or **write** the object state
- Properties can be
  - Declared with all the information hiding levels
  - Be static
  - Be abstract
  - Be overridden

Properties

```
public class Circumference {
  private int x;

  public int X {
    get { return x; } // Read Only
  }
                    // Read and Write
  public uint Radius  { get; set; }
                    // Read Only
  public int Y { get; private set; }

  public void Move(int relx, int rely) {
    x += relx;
    Y += rely;
  }
}
```

Modularity

- The act of partitioning a program into individual components (modules) in order to reduce its complexity to some degree [Booch, 1996]
- Each **module** could be compiled separately, and may have connections with other modules (**reutilization**)
- A module can be
  - Functions and methods
  - Classes and types
  - Namespaces and packages
  - Components
  - …

Coupling and Cohesion

- Bertrand Meyer states five **criteria**, **rules** and **principles** about modularity [Meyer, 2000]
- They are commonly summarized in two:
  - **Coupling**: Degree of interconnectedness of modules
  - **Cohesion**: Degree of connectivity among the elements of a single module
- In software design, **loose coupling** and **high cohesion** favor code maintainability and reutilization

Method Overloading

- **Method overloading** allows creating several method implementations with the same name
- In C#, each overloaded method must differ from each other in at least one of the following features
  - The number of its parameters
  - The type of one of its parameters
  - The argument passing mechanism of one of its parameters (value, `ref` or `out`)
- The last point is not applicable for Java

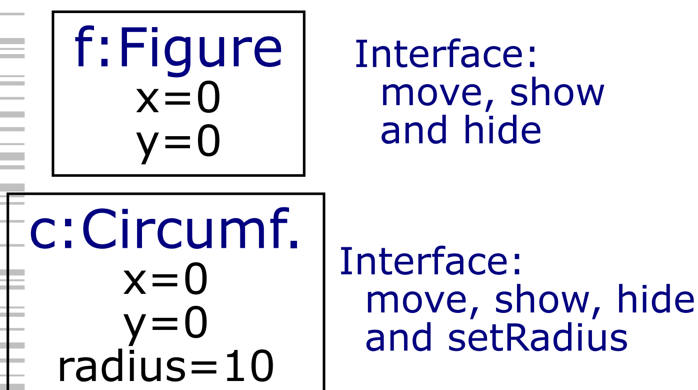
Operator Overloading

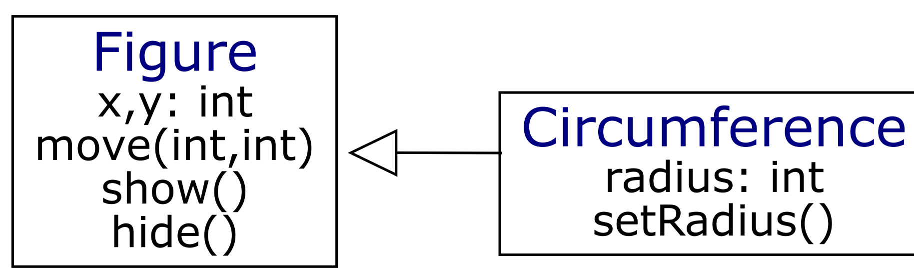
- **Operator overloading** allows operators to have <u>different meanings depending on their parameter types</u>
- C# provides operator overloading, including prefix and postfix `++` and `--`, `[]` (*indexers*), and explicit (cast) and implicit conversions
- Even though C# provides operator overload, it is not widely used

Why?

Inheritance

- Inheritance is a **code reutilization technique** (without considering polymorphism)
- The **state** of derived instances is made up of the union of the fields in the base and derived classes
- The set of messages to be passed (**interface**) to a derived instance is the union of the messages in the base and derived classes

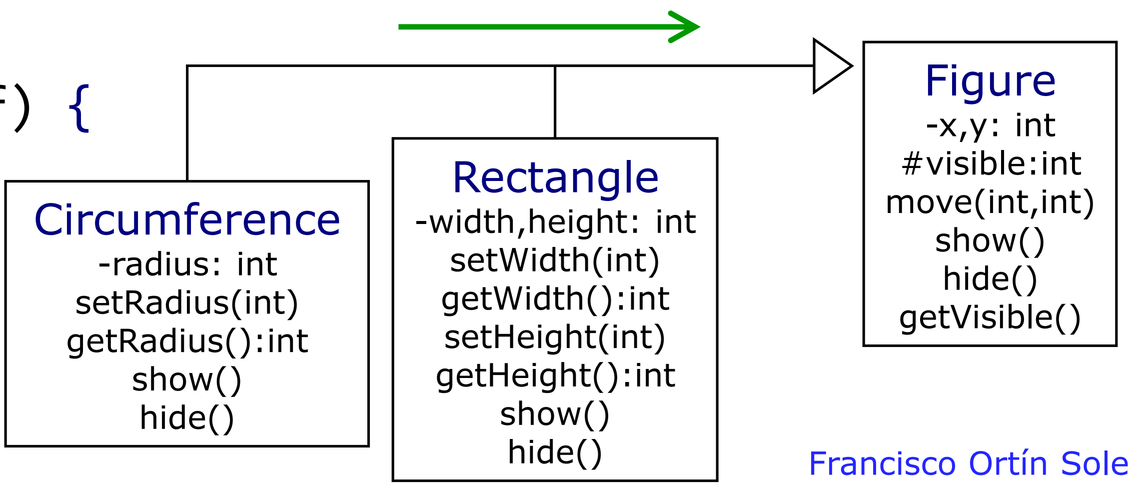




Polymorphism

- Polymorphism is a **generalization** mechanism that allows a general abstraction to represent more specific abstractions
  - The general type represents many types (*poly morphism*)
- Implies a subtyping mechanism (implicit conversion)
  - **Derived** (specific) **references promote** (are implicitly converted) **to base** (general) **references**

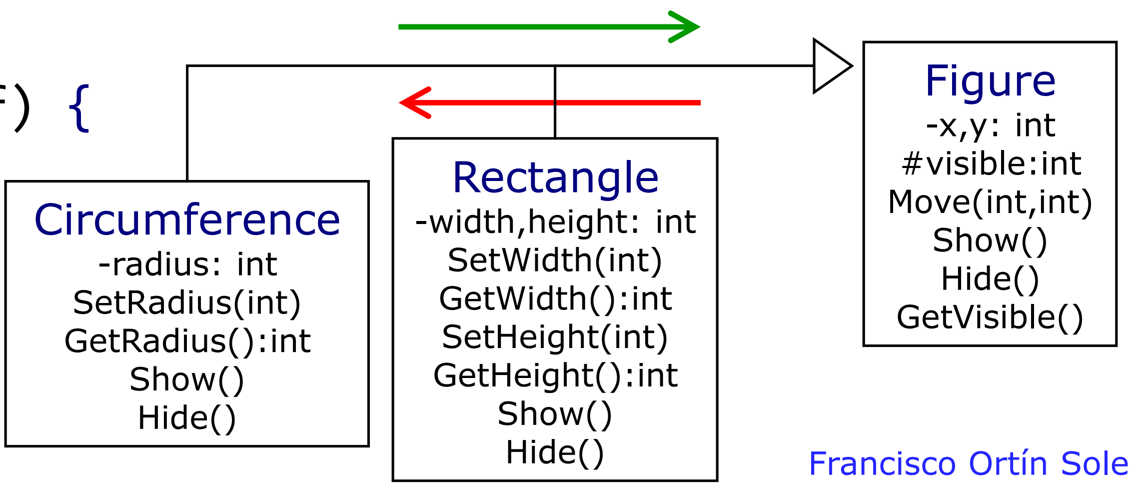
```
void Method(Figure f) {
  …
}
```



Polymorphism

- Only the messages in **`Figure`** can be passed to `f`
- Since `f` could be a **`Circumference`** or a **`Rectangle`**, it is meaningless to pass the **`getRadius`** or the **`getWidth`** message
  - Therefore, downward conversion must be explicit (cast)
  - It can throw an **`InvalidCastException`** if the object has not the expected type
  - C# offers the **`is`** and **`as`** operators to know the dynamic type of a reference

```
void Method(Figure f) {
   …
}
```

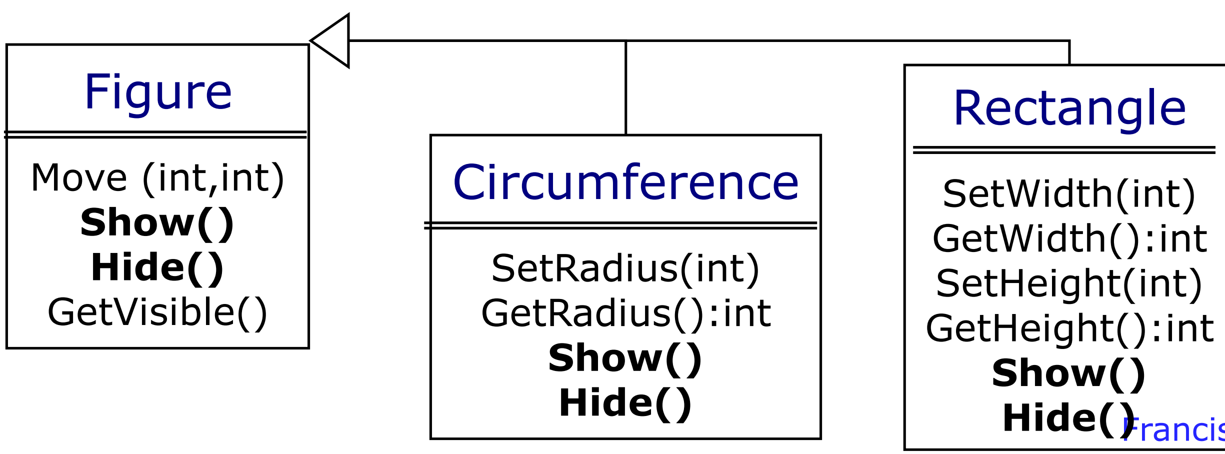


Dynamic Binding

- Dynamic binding is a **specialization** mechanism: the implementation of derived methods can be **specialized** in the derived classes (e.g., `Show` and `Hide`)
- What would happen in the execution of the following code? What would be the actual method called?

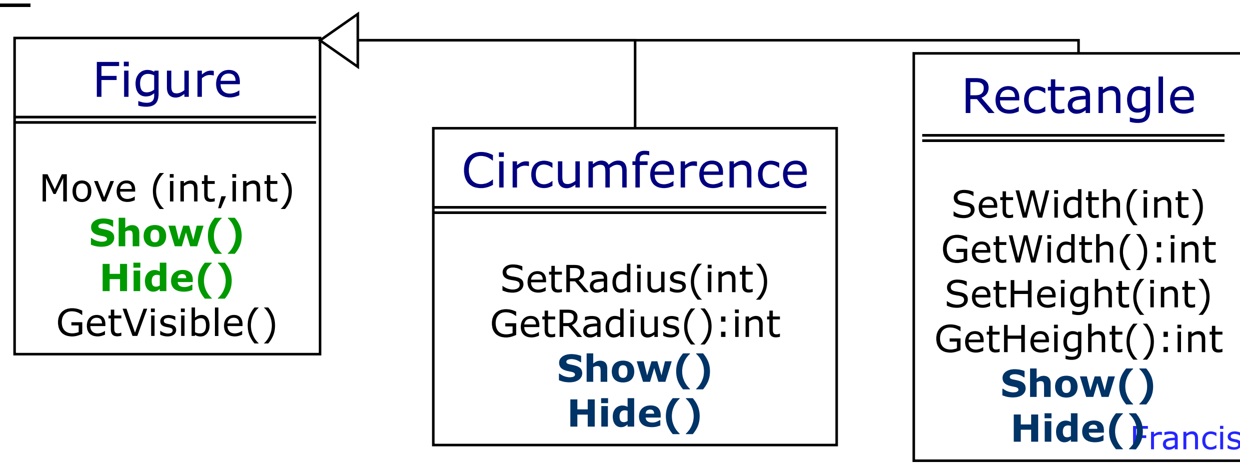
```
void Show(Figure f) {
  f.Show();
}
```

- The method to be called must be the one in its dynamic type (instead of its static type) **dynamic binding** should be used



Dynamic Binding

- C# does not provide dynamic binding by default
- To obtain dynamic binding
  - The method in the base class has to be declared as **`virtual`**
  - The derived method that overrides the virtual one must be declared as **`override`**
- If no overriding is required, but methods are named the same, the derived method should be declared as **`new`**
- This is also applicable to properties
- Question: In Java?



virtual
override

Questions

- Is the following C# code accepted by the compiler?

```
String s = "Hello";
Console.WriteLine(s);
Console.WriteLine(DateTime.Now);
Console.WriteLine(new Angle(0));
```

- In case it is accepted,
  - Why is the implementation correct?
  - What is the positive outcome?
  - How do you think the `WriteLine` method must have been developed?

Polymorphism = Maintainable Code

- There are multiple implementations of **`WriteLine`**
  `WriteLine(int)`, **`WriteLine(char)`**, **`WriteLine(String)`**… **`WriteLine(Object)`**
- In .NET 1.0, the generic way to handle *any* object is using the **`Object`** type
  - .NET 2.0 added generics, providing a safer type mechanism to handle *any* object
- Therefore, **`WriteLine`** is able to show any object in the console
  - It can even show objects defined after the implementation of **`WriteLine`**!
- What method in **`Object`** is **`WriteLine`** using?

The Object Class

Object

Equals(Object):bool
GetHashCode():int
GetType():Type
<u>ReferenceEquals(Object,Object):bool</u>
ToString():String

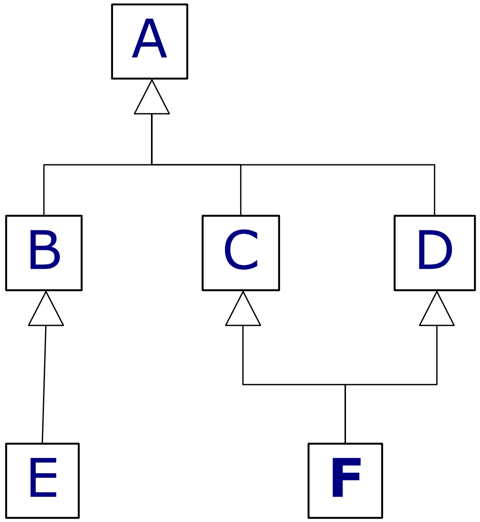
- **`ToString`** returns a string that represents the current object
- <u>It is implemented once, and used in many different scenarios:</u>
  - Show objects in the console
  - Show unhandled exception messages
  - Show the elements in a **`ComboBox`**
  - When the concatenation (+) operator is used
  - …
- By default, **`ToString`** returns a string with the name of the type
- Therefore, it should be overridden in most cases

Abstract Classes and Methods

- When we need a class to provide a **message**, but it cannot be implemented (it is too general), that message is declared as an <u>abstract method</u>
  - C# provides the **`abstract`** keyword
  - The method is not implemented (it is simply a message)
- Abstract methods should be overridden
  - **`abstract`** methods cannot be declared as **`virtual`** (opposite to C++)
  - Remember to state **`override`** when overriding
- A class with at least one abstract method should be declared as an **`abstract` class**
- There could be abstract classes without any abstract method (for code reuse purposes)

Multiple Inheritance

- A programming feature that allows <u>directly</u> **inheriting from more than superclass**
- The derived class inherits <u>all the members</u> in <u>all the superclasses</u>
- C++, Eiffel and Python provide multiple inheritance
  - Java and C# only provide single inheritance



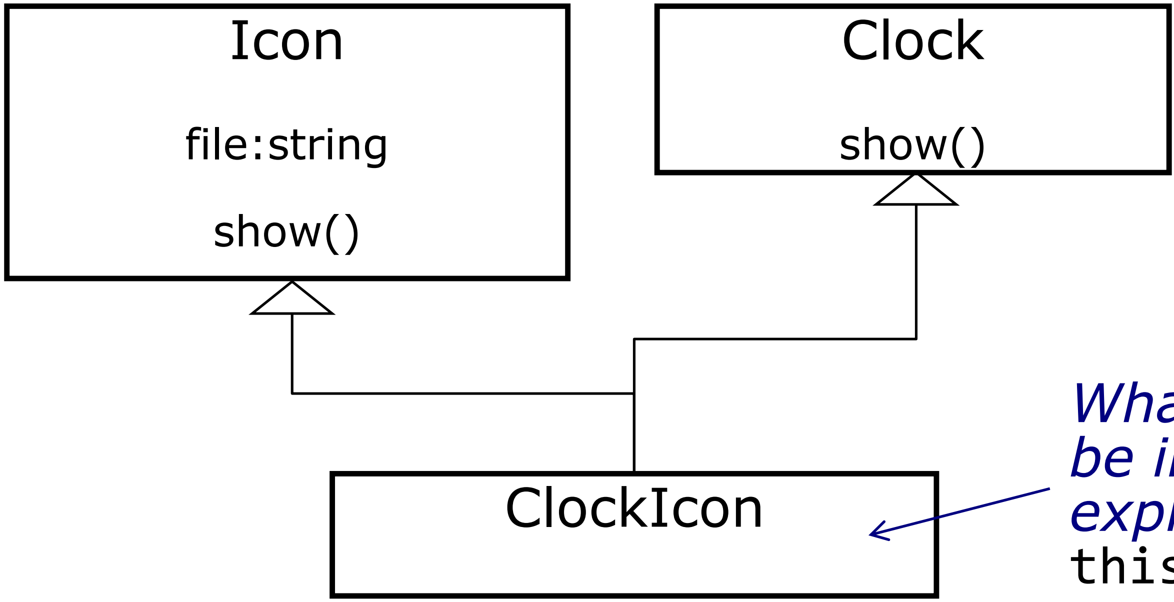
members(instance(F)) =

members(F) ∪
members(C) ∪
members(D) ∪
members(A)

Multiple Inheritance

- Multiple inheritance produces two conflicts:
  1. **Name conflict**: Produced when two superclasses have different members with the same identifier

     Accessing this member in the derived class is ambiguous

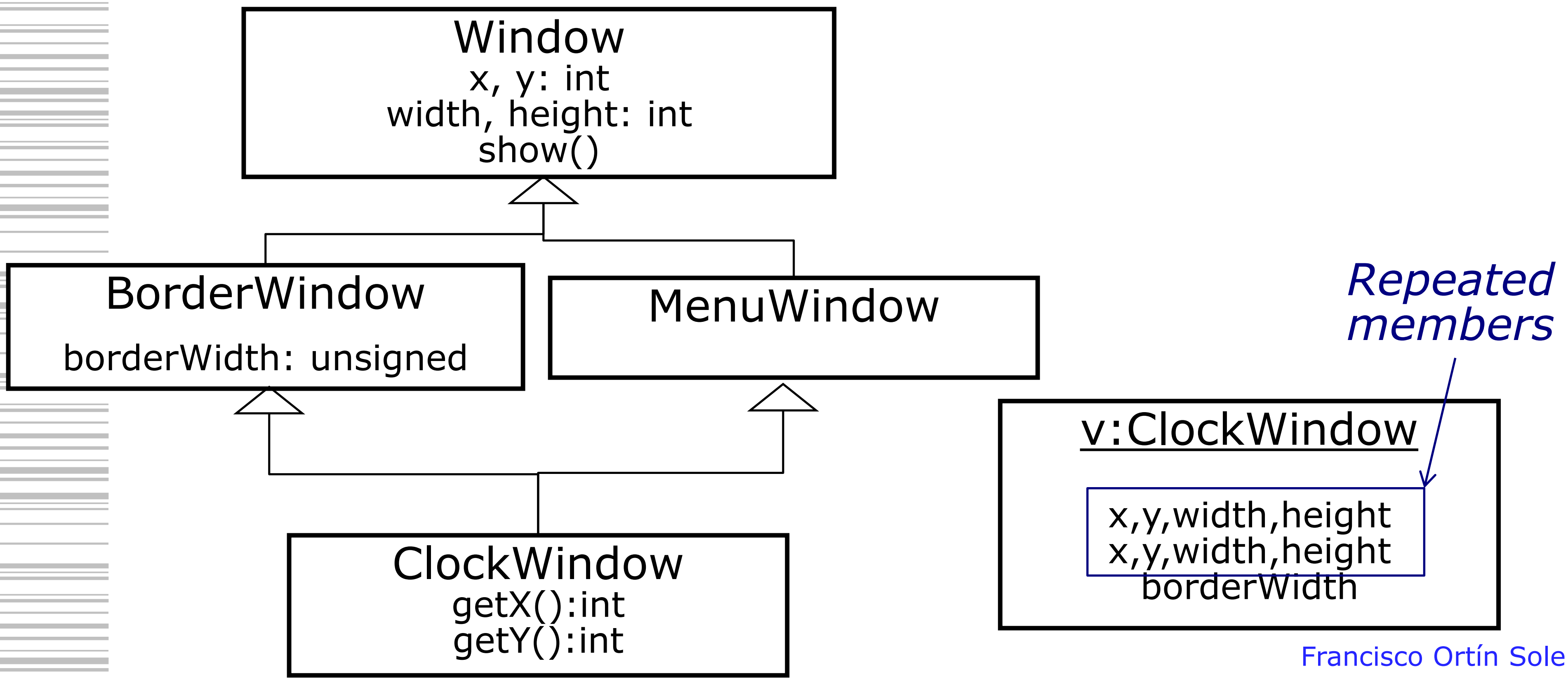


What method would be invoked in the expression?
`this.show()`

Multiple Inheritance

2. **Repeated inheritance** (diamond problem): Produced when a class derives more than once from the same superclass

   Repeated members are inherited



Interfaces

- Due to the conflicts of multiple inheritance its usage was analyzed
  1. In most cases, multiple inheritance was not used as a code reutilization mechanism, i.e. as **inheritance**
  2. It was mostly used as a multiple generalization technique, i.e. "multiple" **polymorphism**
- For the (less common) first case, **composition** is an alternative code reutilization technique
- For the (most common) second case, some single-inheritance languages included the concept of **interface as a type** ("multiple" polymorphism)

Interfaces

- In many occasions, we need a type to be a subtype of <u>two or more super-types</u>
  - What we are looking for is actually <u>"multiple" polymorphism</u>
- As we have learned, an **interface** is <u>the set of **messages** a class provides</u> to their clients (public)
- <u>In C#</u> (and Java), this concept is offered **as a type**
- Interfaces provide **multiple subtyping** (polymorphism)
  - A **`class`** or **`interface`** <u>may derive</u> (implement) from **one or more** **`interface`**s

Seminar 1

- Seminar 1
  - Polymorphism and Dynamic Binding

Dynamic Error Handling

- A compiler is <u>not able to detect every possible error in a program</u>
  - Many errors depend on the application <u>dynamic context</u> (runtime conditions)
  - <u>Examples</u>: array index out of range, out of memory, division by zero, unfulfilled preconditions…
- For this reason, many programming languages provide a mechanism to handle runtime errors
  - Historically, this runtime error handling was done with <u>*ad hoc* code</u>

Exception Handling Objectives

- A mechanism to control runtime errors should be
  1. **Reusable**: We can reuse the code, implementing <u>different error handlers</u> in different scenarios
  2. **Robust**: The mechanism should allow <u>forcing</u> the programmer to write code that <u>handles</u> the runtime <u>error</u> (e.g., what happens if a file cannot be opened?)
  3. **Extensible**: Add <u>new kind of errors</u> and extend the existing ones

Exceptions

- **Exceptions** are objects holding essential information about an exceptional situation occurred at runtime
- Exception handling is based on **separating**
  - Detecting the error and throwing the exception (provider)
  - Different code sections that handle the exception (clients)
- In C#
  - All the exceptions are *unchecked* (i.e., **`RuntimeException`** in Java)
  - Exceptions thrown by a method are not specified (i.e., **`throws`** en Java)

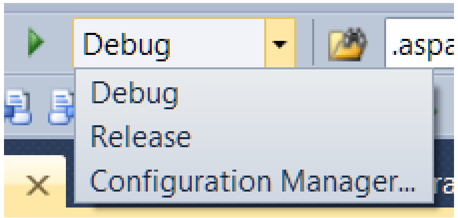
Assertions

- **Assertions** are conditions (predicates) that must be true in the correct execution of a program
  - If they are false, application execution terminates
- Assertions **must not be used to handle runtime errors**
  - They are neither reusable nor extensible
- So, when do we use assertions? [*Steve McConnell, 2004*]
  - To check those conditions that should never happen at runtime (postconditions, invariants…)
  - If the condition is false, it is due to a **programming error** that must be corrected
  - They should be transparently removed in the release version (upon deployment)

Assertions in C#

- Provided by the **`Debug`** class in the **`System.Diagnostics`** namespace

  `Debug.Assert(bool condition:bool, string message)`
- They are enabled in the *Debug* compilation mode…
- …and disabled in *Release* mode



Generics (Genericity)

- **Generics** (genericity) allows writing abstraction (data structure and algorithm) patterns valid for multiple types
- The two main **benefits** provided are
  - Better robustness (compile type error detection)
  - Better runtime performance
- **C# 2.0** permit the generic implementation of
  - Classes
  - Structs
  - Methods
  - Interfaces
  - Delegates
- The .NET platform 2.0 (virtual machine) has been modified to natively support generics

Generic Methods

```
class Generics {
  public static T ConvertReference<T>(Object reference) {
    if (!(reference is T))
      return default(T); // default value of T type (0 for int)
    return (T)reference;
  }

  public static void Main() {
    Object myString = "hello", myInteger = 3;
    // Correct conversions
    Console.WriteLine(ConvertReference<String>(myString));
    Console.WriteLine(ConvertReference<int>(myInteger));
    // Wrong conversions
    Console.WriteLine(ConvertReference<int>(myString));
    Console.WriteLine(ConvertReference<String>(myInteger));
}
```

Generic Classes

```
class GenericClass<T> {
    private T field;
    public GenericClass(T field) {
        this.field = field;
    }
    public T get() {
        return field;
    }
    public void set(T field) {
        this.field = field;
    }
}
class Run {
  public static void Main() {
    GenericClass<int> myInteger = new GenericClass<int>(3);
    Console.WriteLine(myInteger.get());
    GenericClass<string> myString = new GenericClass<string>("hello");
    Console.WriteLine(myString.get());
    }
}
```

Bounded Generics

- For the following generic method

```
T MyMethod<T>(T parameter)  { ... }
```

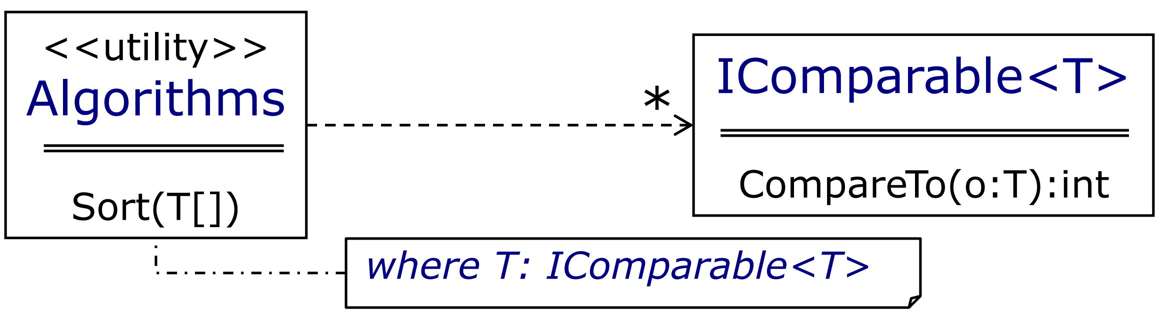
  or any generic class

```
class MyClass<T> {
...
}
```

- What messages can be passed to the generic variables (`T` in the code above)?
- Only those messages in `Object` can be passed
  - By default, generic variables are `Object`s
- Then, how could we implement a generic `Sort` method?

Bounded Generics

- **Bounded** (constraint) **generics** allows making generic parameters more specific
  - And hence, less general (i.e., bounded)
  - Thus, more messages can be passed
- For example, the **`Sort`** generic method below sorts any array of *elements that implement the* ***IComparable<T>*** *interface*




```
static T[] Sort<T>(this T[] vector) where T : IComparable { … }
```

- Question: What's the difference with?

```
static IComparable[] Sort<T>(this IComparable[] vector) { … }
```

Seminar 2

- Seminar 2
  - Generics

Generics in C#

- The .NET Framework makes extensive use of generics
- From now on, we will use the following generic collections
  - `IEnumerable<T>`
  - `IList<T>` and `List<T>`
  - `IDictionary<TKey, TValue>` and `Dictionary<TKey, TValue>`
- <u>Question</u>: Corresponding classes in Java?
- They are explained in the **mandatory activities**
- Their knowledge and use is a <u>**required prerequisite to understand the following theory, seminar and laboratory classes**</u>

Generics in Java

- Java **1.5+** included generics in the programming language
- However, the Java virtual machine (JVM) does **not** support generics
  - The Java compiler translates generic types (i.e., *T*) to `Object`
  - Therefore, at the JVM level, **polymorphism is used** instead
- This technique has many **limitations**
  - Less opportunities for runtime performance optimizations
  - Primitive types can not be used as generic types (e.g. an `ArrayList` of `int`)
  - Cannot create new instances of generic types (`new`)
  - Cannot use `static` generic types (e.g., `static T field;`)
  - Cannot cast or `instanceof` generic types
  - Cannot create arrays of generic types
  - Cannot use overload with differently instantiated generic types

Type Inference

- **Type inference** (a.k.a., type reconstruction) refers to the automatic deduction of the type of an expression
- The less information provided by the programmer (e.g., in variable declaration), the **more powerful** type inference is
- For example, the following ML code (F#) infers the type of `f` to be `int f (int a, int b)`

```
let f a b =
      a + b + 100
```

Type Inference in C#

- C# provides **type inference** in the following three main scenarios
  1. Generic methods
  2. Implicitly typed local variables (`var`)
  3. Lambda functions (next unit)

Type Inference in Generic Methods

- In many cases, generic types in a generic method do not need to be specified upon invocation

```
static void Swap<T>(ref T lhs, ref T rhs) {
  T temp;  temp = lhs;
  lhs = rhs; rhs = temp;
}
static void Main() {
  int a = 1, b = 2;
  Swap(ref a, ref b);
  double c = 3.3, d=4.4;
  Swap(ref c, ref d);
  Swap(ref a, ref d); // Compiler Error
}
```

generics/inference

Implicitly Typed Local Variables

- C# allows avoiding specifying the type of local variables upon declaration
  - `var` should be used instead of the variable type
  - An expression should be assigned to the variable in the declaration

```
var vector = new[] { 0, 1, 2 };  // vector is int[]
foreach(var item in vector) {    // item is int
   ...
}
```

- It is useful when
  - Types have long names (due to generics)
  - It is not easy to know the type of an expression (e.g., LINQ)
  - There is not an explicit type (i.e., anonymous types)

Computer Science
Engineering School

Software
Engineering

Programming Technologies and Paradigms

The Object Oriented Paradigm

Francisco Ortín Soler

University of Oviedo