

Virtual Execution Environments

2021

Week 3

Types at Runtime

Logging Arrays

Arrays

- Arrays are Sequences of elements, e.g. n1, n2, ...
- Sequences in dotnet are instances of **IEnumerable** ↔ Iterable in Java
⇒ Arrays are instances compatible with IEnumerable.

Behavior of Logging arrays:

- ⇒ Traverse elements (e.g. **foreach**) and print each element
- ⇒ To perform a **foreach** we only need a **IEnumerable**

Check Compatibility

- `o is IEnumerable` ⇔ instanceof Java

Using Reflection API with instances of Type

- `IsSubclassOf` – only for Types representing **classes**

- Does not work here:

```
o.GetType().IsSubclassOf(typeof(IEnumerable))
```

- `IsAssignableFrom` - for any kind of Type

```
typeof(IEnumerable).IsAssignableFrom(o.GetType())
```

Given two objects at runtime e.g. o1 and o2 !!! We cannot use 'is'

Check Compatibility !!!! OVERHEADS 2 x!!!!

```
string output = o is IEnumerable
                ? Inspect((IEnumerable) o)
                : Inspect(o);
```

isinst or casclass:

1. pop ref from Stack
2. Get the Type of that ref
3. Check if that Type is the desired Type (i.e. IEnumerable)
 1. Yes -> return the ref
 2. Not -> track hierarchy (including interfaces) and repeat this procedure until reach object.
4. If not 3.1

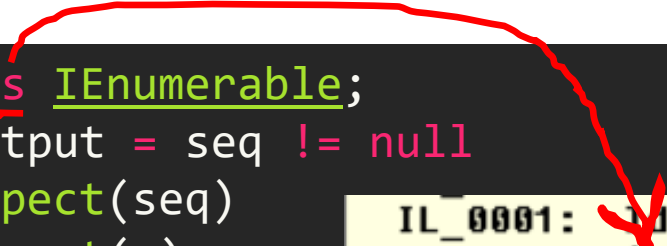
2x

```
IL_0001: ldarg.1
IL_0002: isinst [System.Runtime]System.Collections.IEnumerable
IL_0007: brtrue.s IL_0012
IL_0009: ldarg.0
IL_000a: ldarg.1
IL_000b: call instance string Logger.Log::Inspect(object)
IL_0010: br.s IL_001e
IL_0012: ldarg.0
IL_0013: ldarg.1
IL_0014: castclass [System.Runtime]System.Collections.IEnumerable
IL_0019: call instance string Logger.Log::Inspect(class [System
IL_001e: stloc.0
IL_001f: ldarg.0
IL_0020: ldfld class Logger.IPrinter Logger.Log::printer
IL_0025: ldloc.0
IL_0026: callvirt instance void Logger.IPrinter::Print(string)
IL_002b: nop
IL_002c: ret
```

Throw **CastClassException** for castclass or **null** for isinst

Check Compatibility 1 x

```
IEnumerable seq = o as IEnumerable;  
string output = seq != null  
    ? Inspect(seq)  
    : Inspect(o);
```



```
IL_0001: ldarg.1  
IL_0002: isinst [System.Runtime]System.Collections.IEnumerable  
IL_0007: stloc.0  
IL_0008: ldloc.0  
IL_0009: brtrue.s IL_0014  
IL_000b: ldarg.0  
IL_000c: ldarg.1  
IL_000d: call instance string Logger.Log::Inspect(object)  
IL_0012: br.s IL_001b  
IL_0014: ldarg.0  
IL_0015: ldloc.0  
IL_0016: call instance string Logger.Log::Inspect(class [System]  
IL_001b: stloc.1  
IL_001c: ldarg.0  
IL_001d: ldfld class Logger.IPrinter Logger.Log::printer  
IL_0022: ldloc.1  
IL_0023: callvirt instance void Logger.IPrinter::Print(string)  
IL_0028: nop  
IL_0029: ret
```

Reflection Overheads

```
Student s1 = new Student(154134, "Ze Manel", 5243, "ze");  
Student s2 = new Student(324234, "Xico", 1234, "xico");  
Student s3 = new Student(763547, "Maria Papoila", 3547, "maria");  
Student[] arr = {s1, s2, s3};
```

```
private string Inspect(IEnumerable seq) {  
    StringBuilder str = new StringBuilder();  
    str.Append("Array of:\n");  
    foreach(object item in seq) {  
        str.Append("\t");  
        str.Append(Inspect(item));  
        str.Append("\n");  
    }  
    return str.ToString();  
}
```

For each Member of an item:
1. Check ShouldLog(member)
2. Yes => GetValue from that Member

The result of **ShouldLog** change from item to item?
R: NO

What is the overhead of **ShouldLog**?
R: In worst case 4 verifications and isinst conversion

```
if(!Attribute.IsDefined(m,typeof(ToLogAttribute))) return false;  
if(m.MemberType == MemberTypes.Field) return true;  
return m.MemberType == MemberTypes.Method && (m as MethodInfo).GetParameters().Length == 0;
```

Minimize calls to ShouldLog

```
MemberInfo[] members = t.GetMembers();  
foreach (MemberInfo member in members) {  
if (ShouldLog(member))  
    ...  
}
```

Only get valid Members

```
private IEnumerable<MemberInfo> GetMembers(Type t)  
{  
    // First check if exist in members dictionary  
    List<MemberInfo> ms;  
    if (members.TryGetValue(t, out ms)) {  
        ms = new List<MemberInfo>();  
        foreach (MemberInfo m in t.GetMembers()) {  
            if (ShouldLog(m))  
                ms.Add(m);  
        }  
        members.Add(t, ms);  
    }  
    return ms;  
}
```

```
Dictionary<Type, List<MemberInfo>> members = new Dictionary<Type, List<MemberInfo>>();
```

Inspect...

```
Dictionary<Type, List<MemberInfo>> members = new Dictionary<Type, List<MemberInfo>>();
```

```
private string LogMembers(object o)
{
    Type t = o.GetType();

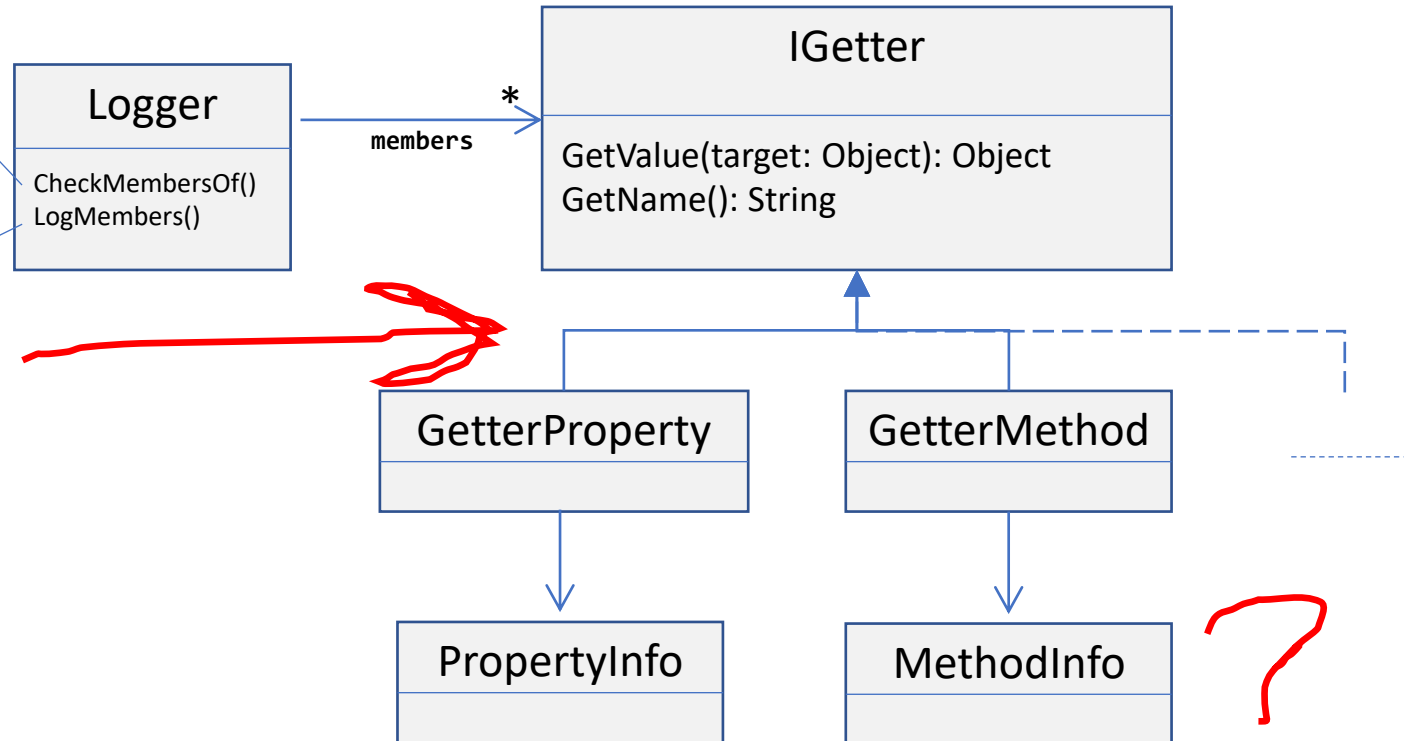
    StringBuilder str = new StringBuilder();
    foreach (MemberInfo member in GetMembers(t))
    {
        str.Append(member.Name);
        str.Append(": ");
        str.Append(GetValue(o, member));
        str.Append(", ");
    }
    if(str.Length > 0) str.Length -= 2;
    return str.ToString();
}
```

```
private object GetValue(object target, MemberInfo m) {
    switch(m.MemberType)
    {
        case MemberTypes.Field:
            return (m as FieldInfo).GetValue(target);
        case MemberTypes.Method:
            return (m as MethodInfo).Invoke(target, null);
        default:
            throw new InvalidOperationException("Non properly ...");
    }
}
```


Logger

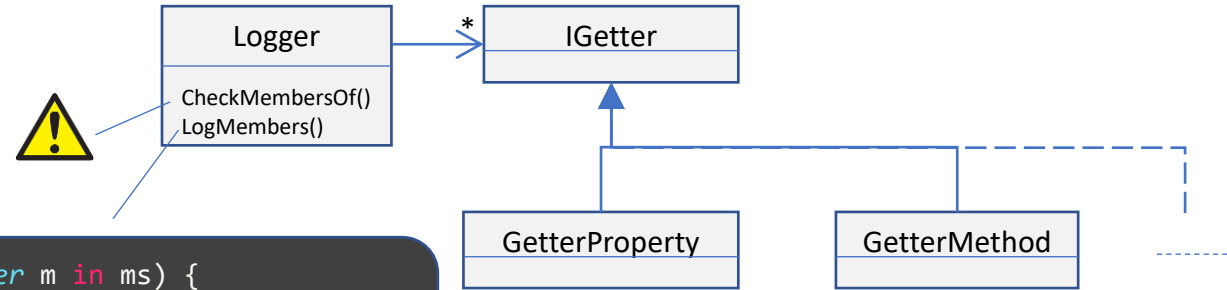
```
static List<IGetter> CheckMembersOf(Type klass) {  
    ...  
    foreach (FieldInfo f in klass.GetFields()) {  
        if (CheckToLog(f)) {  
            ...  
        }  
    }  
    foreach (PropertyInfo p in klass.GetProperties()) {  
        if (CheckToLog(p)) {  
            ...  
        }  
    }  
    ...  
}
```

```
static void LogMembers(Type klass, object target) {  
    List<IGetter> ms = CheckMembersOf(klass);  
    foreach (IGetter m in ms) {  
        Console.WriteLine(" " + m.GetName());  
        Console.WriteLine(": ");  
        Console.WriteLine(m.GetValue(target));  
        Console.WriteLine(",");  
    }  
}
```



Logger e.g. for Student

```
public class Student {  
    public readonly int nr;  
    public readonly string name;  
    public readonly int group;  
    public readonly  
    string githubId;  
}
```

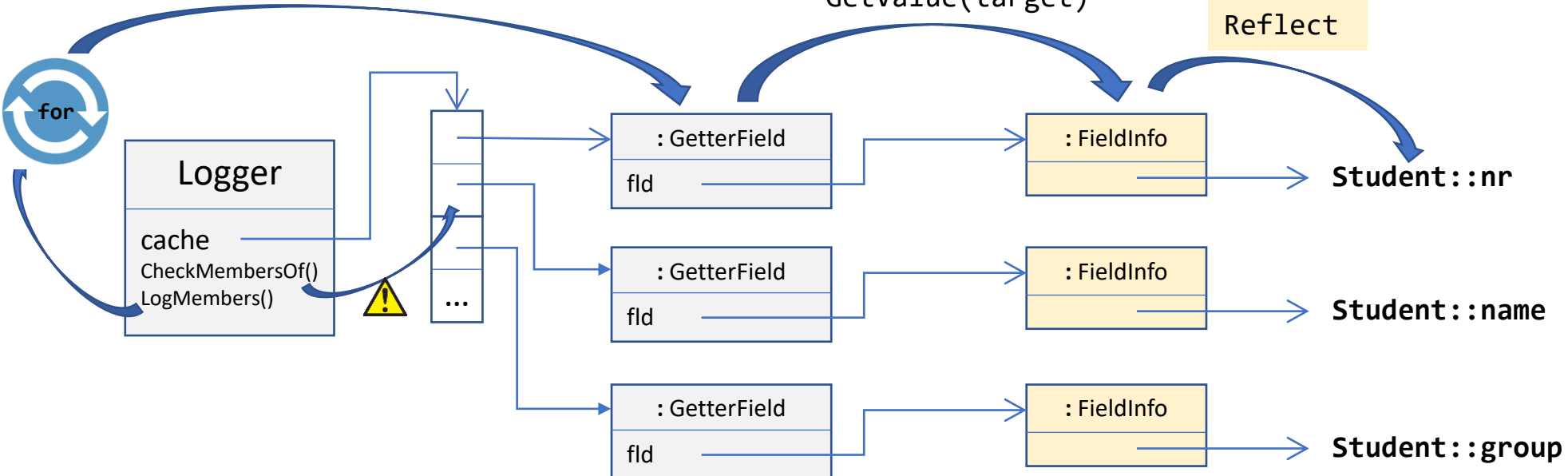


```
foreach (IGetter m in ms) {  
    ...  
    Console.WriteLine(m.GetValue(target));  
    ...  
}
```

Call(target)

GetValue(target)

Reflect



FireMapper

2 formas de aceder a propriedades de classes de domínio:

- Propriedades “simples”: primitivas ou do tipo string.
 - Leem ou escrevem directamente
- Propriedades “complexas”: do tipo de outra classe de Domínio
 - Obtêm o seu valor através de uma outra instância de IMapper

Objectivo (alínea 0 do trabalho 2):

- Criarem uma abstracção que estabelece uma forma única de aceder às propriedades independentemente de serem “simples” ou “complexas”