

Phalanger: Integrating PHP with CLR

Ladislav Prošek, Tomáš Matoušek

July 2006

Introduction.....	2
PHP Object Model	2
Goals.....	3
Consuming CLR Types.....	3
Real Object Wrapping	4
Wrapper Cache.....	4
Instantiation	6
Method Invocation	7
Field and Property Access	8
Subscribing to Events	9
Working with Value Types.....	10
Producing CLR Types	11
Anatomy of a Type	11
Anatomy of an Exported Type.....	14
Extending CLR Types.....	15
Overriding Methods	16
Overriding Properties	16
Invoking Base Constructors	17
References.....	18

Introduction

Phalanger [1] is a PHP language compiler for .NET Framework [2]. It compiles PHP web pages, console applications, and windows applications to verifiable managed .NET assemblies. PHP [3] is a scripting language aimed particularly at rapid development of simple server-side HTML-generating scripts. Its dynamic nature makes the compilation an uneasy task, and also places high demands on the language runtime. Unlike statically typed languages such as C#, which require no runtime support other than the CLR itself, Phalanger comes with a rich set of classes providing various services to running scripts. This document discusses some aspects of compilation and runtime support, in particular those that were added to Phalanger version 2.0 in order to turn it into a full-fledged consumer, producer, and extender of CLR types.

PHP Object Model

Starting with version 5.0, PHP supports a reasonable object model that makes object-oriented programming a viable option. There was already a notion of classes and objects in previous versions, but objects were in fact just associative arrays with functions defined on them. PHP 5.0 added member visibility (public, protected, private), static properties, interfaces, and more.

The following code snippet illustrates how a class is declared in PHP.

```
class Manager extends Employee
{
    public function __construct($level, Manager $reportsTo)
    {
        parent::__construct($reportsTo);
        $this->level = $level;
    }

    public static function GetMaxLevel()
    {
        return self::$maxLevel;
    }

    private $level;

    public static $maxLevel = 5;
}
```

The only members that a class can contain are methods (usually called functions in PHP) and fields (called properties). To avoid confusion, this document strictly uses the CLR terminology. Methods and fields can be either instance or static. Although PHP is often hosted in a multi-threaded environment, it was designed for single-threaded programming model. Therefore static fields are in fact thread-static, as each instance of a script has its own copy which is not shared with other instances.

Constructors are ordinary methods with the special name `__construct`¹. They can be called just like any other methods. After an instance of a class is created using the `new` expression, its

¹ The “old” PHP 4 constructor naming style, in which the constructor method has the same name as the declaring class, is still supported.

constructor method is executed automatically. However, there are no automatic or forced upcalls to constructors of parent classes, so if `Manager::__construct` did not contain the `parent::__construct()` call, the `Employee` class would remain uninitialized. Method formal parameters can be prepended with a class or interface reference, which is called a type hint (see the second parameter of `__construct`). It is a syntactic sugar translated to an instance of precondition check at the very beginning of the respective method.

The important feature that reveals the dynamism of the PHP language is the ability to access fields that have not been declared in the class. Consider the following code.

```
$m = new Manager(1, NULL);
$m->FavoriteMovie = "Matrix";
```

Since there is no `FavoriteMovie` field in `Manager` or any of its parent classes, a new field is added to the instance stored in `$m` at runtime. The fields added in this fashion are public and can be deleted from the instance using the `unset` construct. They are treated just like the declared fields – they are enumerated by the `foreach` statement, dumped, serialized etc.

Goals

The first goal of Phalanger v2.0 is to implement the PHP object model to be entirely compatible with PHP. This also includes many subtleties not mentioned in the preceding overview like e.g. member lookup rules and method overriding rules. The second goal is to make Phalanger interoperable with other .NET languages, which means that the types produced by Phalanger must be easily consumable from, say, a C# program, and vice versa. Phalanger should also be able to extend (and implement) types produced by other languages, work with delegates, subscribe to events and so on.

CLR objects in PHP should behave as PHP objects in all aspects. As an example, consider the following code, in which a runtime field is added to an instance of `System.Xml.XmlDocument`.

```
$doc = new System::Xml::XmlDocument;
$doc->Author = "John Smith";
```

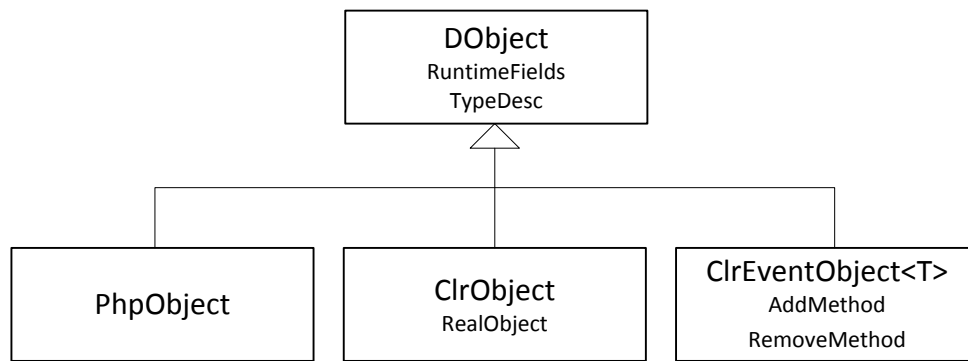
There is no field or property named `Author` in the `System.Xml.XmlDocument` class. According to the PHP rules, a runtime field is added to the instance. Not only must the `Author` field be retrievable back from the `$doc` variable, it must be really bound with the instance so that the following code produces the expected output.

```
$stack = new System::Collections::Stack;
$stack->Push($doc);
echo $stack->Pop()->Author;
```

Consuming CLR Types

By consuming CLR types, we mean being able to instantiate a non-abstract .NET type, call public methods on it, access its public fields and public properties, and subscribe to its public events. In order to support all aforementioned PHP features, instances of .NET types are represented by instances of the `ClrObject` internal Phalanger type instead of the instances themselves. In other words, instances of CLR types are internally always kept wrapped.

Real Object Wrapping



The base abstract `DObject` class represents everything that behaves as an object in PHP. It implements several internal interfaces including `IPhpVariable`, `IPhpComparable`, and `IPhpConvertible`. The two important entities associated with each `DObject` are the table of fields added at runtime (`RuntimeFields`) and a reference to the `DTypeDesc` (`TypeDesc`) that describes the real type of the object². Consider the following snippet, in which a field is read from an object.

```
echo $x->a;
```

This field access is compiled as a call to an operator method that takes an object (typed as `System.Object`) and a name of the field as string. The operator checks whether `$x` is of `DObject` type. If it's not, an error is reported. Otherwise, the operator delegates this call to the `DObject` which tries to retrieve a field/property/event descriptor from its `TypeDesc` and if that fails, from its `RuntimeFields`. Note that all of this code is completely unaware of the fact that `$x` may actually be a CLR object or a PHP object. The difference is in the field/property/event descriptor retrieved from the type descriptor; in this case in their `Get` method, which is finally called on it to obtain the value.

The `ClrObject` class contains a reference to the real object it wraps. Each CLR object that has ever entered the „Phalanger space“ has its wrapper. The wrapping and unwrapping code is emitted to all places where a CLR object may potentially enter or leave the Phalanger space.

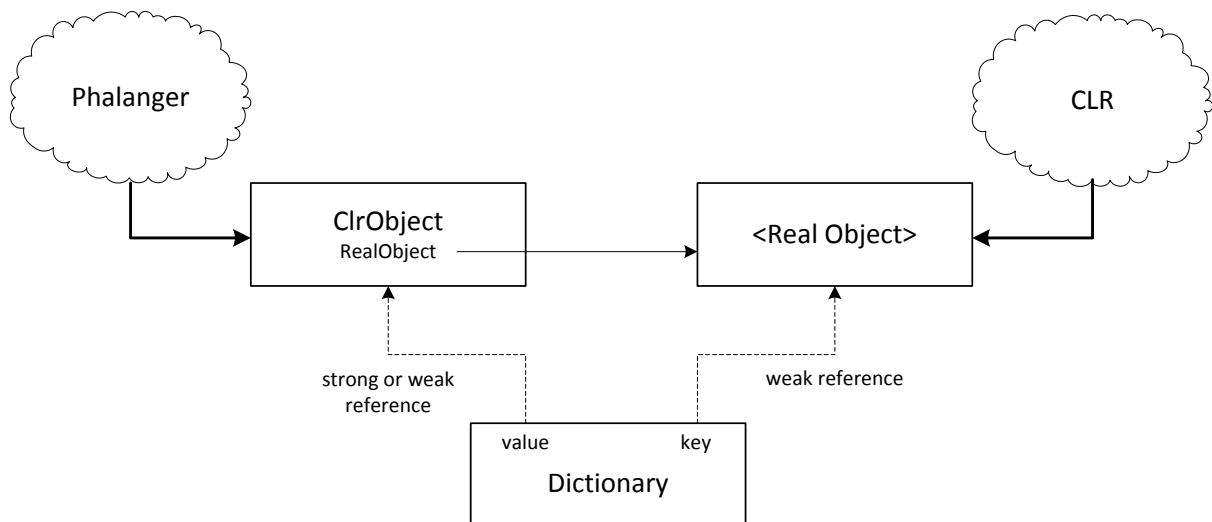
- Overload resolution stubs (executed when a CLR constructor or method is called),
- Field/property access stubs (executed when a CLR field or property is accessed),
- Overriding, implementing, and export stubs (executed when a PHP method or field is accessed from outside Phalanger),
- Delegate stubs (executed when a CLR delegate pointing to a PHP method is invoked).

Wrapper Cache

While Phalanger always holds a reference to an object's `ClrObject` wrapper, passing the object outside Phalanger requires appropriate unwrapping. Apart from extracting the `RealObject`

² Dynamic type descriptors are part of a broader architecture that is not covered by this document. Here they can be regarded simply as a faster alternative to CLR reflection, providing rapid access to type's members.

reference, this may also involve dereferencing a `PhpReference`, which is another type used internally by Phalanger to implement PHP features. Nevertheless, as stated in the goals section, we demand that when the same (in terms of reference equality) object enters the Phalanger space again, it will be wrapped by the same (also in terms of reference equality) `ClrObject`. This could be easily implemented by a static dictionary (e.g. `Dictionary<object, ClrObject>`) if we do not take object life-time into consideration. Such simple dictionary would keep all objects that have ever entered the Phalanger space in the current app domain alive forever, and prevent them from being garbage collected. Therefore a more sophisticated scheme has been implemented.



The real object may be referenced from Phalanger (the cloud on the left) using the `ClrObject` as an intermediary, as well as from outside Phalanger (the cloud on the right) directly. To address the garbage collection issue, we have to introduce weak references. However, referencing only the real object by the dictionary using a weak reference would not be enough as the strongly referenced `ClrObject` would still keep the real object alive. We need to be able to tell when both the wrapper and its real object are eligible for garbage collection, which happens when there are no other references except for the wrapper – real object link and the dictionary references. Basically, a single reference from the Phalanger or CLR cloud must keep the pair alive.

When a real object is initially wrapped, it is added to the dictionary and the key as well as the value are weak. In addition, the value weak reference is set to track resurrections. Now references from the Phalanger cloud keep the pair alive without any additional cost. If there's no reference from the Phalanger cloud but the real object is referenced by CLR, the `ClrObject` eventually becomes eligible for finalization. Its finalizer detects the fact that the real object is still alive by checking the corresponding entry in the dictionary, and resurrects itself by replacing the resurrection-tracking weak reference by a strong reference in the dictionary entry's value. If the real object is no longer alive, the `ClrObject` lets itself die too. The dictionary periodically traverses all pairs and removes those that are dead. In addition, strong references are converted back to resurrection-tracking weak references during these sweeps, effectively scheduling another finalizer execution.

The periodic checks and resurrections performed by the `ClrObject`'s finalizer hurt performance and should be subject to further research.

Instantiation

When a particular CLR type is instantiated in a given app domain for the first time, a stub (dynamic method) is generated. This stub has a unified signature so that a delegate pointing to it can be added to internal tables and reused when the same type is instantiated later on. The stub employs overload resolution to find a suitable constructor (see the part about methods for details). Constructor arguments that have been pushed to an internal argument stack (the `PhpStack`) are popped by the stub and passed as CLR arguments to the target type constructor.

The following example is a C# approximate of the constructor stub generated by current version of Phalanger for the `System.Xml.XmlDocument` class.

```
static object System_Xml_XmlDocument_ctor(object _, PhpStack stack)
{
    switch (stack.ArgCount)
    {
        case 0:
            stack.RemoveFrame();
            return ClrObject.Wrap(new XmlDocument());

        case 1:
            break;

        default:
            PhpException.InvalidArgumentCount(
                "System::Xml::XmlDocument", null);
            break;
    }

    object arg1 = stack.PeekValueUnchecked(1);
    stack.RemoveFrame();

    bool success;
    XmlImplementation implementation1 =
        Convert.TryObjectToClass<XmlImplementation>(arg1, out success);
    if (success && stack.AllowProtectedCall)
    {
        return ClrObject.Wrap(new XmlDocument(implementation1));
    }

    XmlNameTable table1 =
        Convert.TryObjectToClass<XmlNameTable>(arg1, out success);
    if (success)
    {
        return ClrObject.Wrap(new XmlDocument(table1));
    }

    PhpException.NoSuitableOverload("System::Xml::XmlDocument",
        ".ctor");
    return null;
}
```

Please note that the newed up real objects are immediately wrapped by new `ClrObjects`. Also note that one of the three target type's constructors has protected visibility, which means that an extra check is performed by the stub to see if it is allowed to call the overload. This becomes important when a PHP class derives from a CLR type (see the chapter about extending).

Method Invocation

Dynamic methods are generated at runtime also for methods invoked on CLR types. The PHP language does not support method overloading and there is currently no way how to provide a hint for the compiler as to in which overload the user is interested. Thus the principal task of a stub is to choose the most appropriate overload of the target method based on the number and types of the supplied actual arguments.

Each stub contains a switch which essentially filters out overloads with the number of formal parameters that is smaller or greater than the number of actual arguments supplied to the call. Afterwards, the overloads with the correct number of parameters are tested one-by-one by trying to convert arguments to their parameter types. The first overload, for which all arguments are convertible to its parameter types, receives the invocation. The conversions must constitute a sensible trade-off between dynamism and selectivity. PHP can convert any type to just about any other type but if the conversion routines behaved like this, inappropriate overloads would have been called. For instance, every object in PHP is implicitly convertible to a number, which yields either 0 or 1 depending on whether there are any fields in the object. The conversion routines involved in overload resolution must suppress these heavy-weight conversions.

Phalanger fully supports calling methods with `ref` and `out` parameters, because PHP contains the notion of by-reference argument passing. If an argument is about to be passed by reference, it is pushed to the stack wrapped by a `PhpReference`. The value is unwrapped, converted to the corresponding parameter type, stored to a local variable, and the address of the local is passed to the target method. When the method returns, the (possibly updated) local is converted back to Phalanger type and stored back to the `PhpReference`.

Methods with a `params` parameter, i.e. an array parameter decorated by the `ParamArrayAttribute`, have slightly more complicated stubs. Consider the following set of overloads.

```
void Foo();
void Foo(int x1);
void Foo(int x1, int x2);
void Foo(int x1, int x2, int x3);
void Foo(params object[] args);
```

The last `params` overload can become the most appropriate overload for every argument count except for zero. The generated stub for this set of overloads has to consider it in every `switch` branch because if the conversions to `int` fail, the last overload is a fallback. Moreover, there must be a default case containing a loop, in which all actual arguments are converted to the `params` array. Finally, the `params` overload can also be called by explicitly passing an array, so the `switch` branch for exactly one argument on the stack contains the following.

POP arg1 from stack

IF arg1 is convertible to int, call the Foo(int) overload

ELSE IF arg1 is convertible to object, create new object[] { arg1 } and call the Foo(object[]) overload

ELSE IF arg1 is convertible to object[], call the Foo(object[]) overload

ELSE error

Field and Property Access

As well as other .NET languages, Phalanger uses the same syntax to access fields (storage locations inside objects or types) and properties (pairs of methods used to get and set a logical property of an object or type, usually backed by a non-public field). However, unlike e.g. C#, constants are accessed using a different syntax as illustrated in the following snippet.

```
echo System::Environment::$TickCount;  
echo System::Int32::MaxValue;
```

The dollar character is present when referring to a static property while it is missing when referring to a constant. The possibility to declare a static field and a constant both of the same name in PHP is therefore an issue for various reasons. For example, if the type is to be used from different languages, one of the members has to be renamed. Another problem reveals when using .NET CodeDom for generating PHP source code. The CodeDom represents static field access exactly in the same way as constant access, which complicates code generation a bit. Time consuming reflection has to be used every time a static access is generated to determine whether it is a field or a constant.³

Similarly to methods, Phalanger uses run-time generated dynamic methods to read and write properties and fields of CLR objects. The two types of delegates that point to the generated stubs are declared as follows.

```
public delegate object GetterDelegate(object instance);  
public delegate void SetterDelegate(object instance, object value);
```

The `GetterDelegate` receives a reference to an object instance and returns the current value of the respective property or field. The `SetterDelegate` is given the instance and a new value, which it sets to the property or field. The instance is a `null` reference for static properties and fields. The lack of overloads makes the property and field access stubs much simpler than method stubs. The following IL is the getter stub of `System.Environment.TickCount`.

```
.method static object Get_System_Environment_TickCount (object instance)  
{  
    L_0000: call int32 [mscorlib]System.Environment::get_TickCount()  
    L_0005: box int32  
    L_000a: ret  
}
```

³ When the type whose static field or constant is accessed cannot be reached by reflection (because it is for example also being generated by CodeDom), the generator has to guess. This is not the only design flaw of CodeDom, which is regrettably far from being universal and language-independent.

Constants of CLR objects have no stubs as their value is immutable. Instead, the value is directly stored in internal tables. Reading such a constant results in simply returning the value that is cached in the tables.

Subscribing to Events

The last kind of type members that still need to be covered are events. Events are nothing but metadata encapsulating (usually) two methods declared on the type and containing the logic for adding and removing handlers to and from the delegate's invocation list. Since in a dynamic language we are in general unable to determine at compile time whether a += or -= operator is applied to a field/property or an event and as these operators are only shortcuts for expressions $\$x = \$x \pm \$y$, we decided to introduce another syntax for working with events.

```
$domain = AppDomain::$CurrentDomain;
$domain->UnhandledException->Add(
    new UnhandledExceptionHandler("MyHandler"));

// ...

$domain->UnhandledException->Remove(
    new UnhandledExceptionHandler("MyHandler"));
```

Using the syntax of a field/property access, the event is read from its declaring type (for static events) or instance (for instance events). The result is an object with two methods on it: `Add` and `Remove`. Internally this object is represented by an instance of `ClrEventObject<T>`, where `T` is the type of the event, always derived from `System.Delegate`. Please refer to the hierarchy diagram at the beginning of this chapter. The trick is in the fact that a `ClrEventObject<>`'s `RealObject` is the event object itself so calling `Add` on the event is routed to the wrapper that passes the call to the event's `add` method, a delegate to which is held in its field. In another words, events are represented by artificial „event objects“, which in contrast to `ClrObjects` do not wrap references to real objects, but delegates to the corresponding `add/remove` pair of methods.

In Phalanger, delegates can be constructed to point to PHP functions or PHP methods. Consistent with the notion of callbacks used by some of the PHP built-in library functions, we permit the following delegate constructor arguments.

- A string containing the name of the target function,

- An array containing the name of the target type in its 0th element and the name of the target method in its 1st element (the method must be static),

- An array containing the target object instance in its 0th element and the name of the target method in its 1st element (the method may be either instance and static).

Currently there is even an implicit conversion from string or two-element array to any delegate type, so it is possible to write the following code.

```
$domain->UnhandledException->Add("MyHandler");
```

When a delegate is being created, a dynamic method with the matching signature is generated. The method performs necessary checks and conversions (this is one of the points where data may enter or leave the Phalanger space), which are basically the reverse of what the method stubs do. A delegate pointing to the stub is then wrapped by a `ClrObject` and returned. No syntactic sugar is currently provided for delegate invocation; users have to explicitly call the `Invoke` method.

Working with Value Types

Throughout Phalanger, we keep references to real CLR objects in variables of the `System.Object` type, which means that we box value types as they enter the Phalanger space and we unbox them when they leave. For most cases, handling the value types this way exhibits an expected behavior. Before calling a method or accessing a field of a boxed value, we use the `unbox` IL instruction to adjust the instance pointer. Therefore the method correctly works upon the boxed data and the writes to the field really change the boxed data stored on the heap. If, however, the value is embedded in another type (no matter if value or reference), updates to the value's data are not reflected by the original storage. The following example should shed some light on the issue. Consider a structure and a class declared in C# as follows.

```
public struct MyStruct
{
    public int Field;
}

public class MyClass
{
    public MyStruct Structure;

    public void Dump()
    {
        Console.WriteLine(Structure.Field);
    }
}
```

This C# code works as expected and prints 42.

```
MyClass c = new MyClass();
c.Structure.Field = 42;
c.Dump();
```

Unfortunately the "equivalent" code written in PHP and run under Phalanger prints 0.

```
$c = new MyClass;
$c->Structure->Field = 42;
$c->Dump();
```

The structure is read from the class, boxed, and the field is modified on the heap. There is no write-back so the original structure, which is embedded in the class instance, is not affected. This issue should be solved in upcoming versions of Phalanger. A new class derived from `DObject` should probably be introduced and returned by the `->` operator when the field is an embedded structure.

This `DObject` descendant would refer to the real object by a pair consisting of a reference to its containing instance and a field designation.

Producing CLR Types

By producing CLR types, we mean being able to declare a class in PHP that is consumable by other .NET languages. Phalanger can currently produce only classes and interfaces. The PHP language does not support declaring structures (value types), delegates, and nested types. We also lack the possibility to declare events in PHP at the moment.

Anatomy of a Type

Types compiled by Phalanger contain considerable amount of internal members that support the Phalanger runtime. In addition, field types as well as method signatures are very special and optimized for intra-PHP use, thus generally unconsumable by anything else than Phalanger. Making a type consumable requires adding appropriate stubs, which has a significant impact on the size of the resulting assembly caused by the extra IL and metadata. That's why types are not exported by default, but only when explicitly decorated by the `Export` attribute.⁴ We shall now provide a quick overview of the internal helper members that can be found in Phalanger-generated types. Consider the following PHP class declaration.

```
class A
{
    const c = 3.14;

    public $a = "test";
    protected static $b = TRUE;

    public function __construct(A $t = NULL)
    {
        if ($t != NULL) $this->a = $t->a;
    }

    private static function Foo()
    {
        return array(0 => "zero", 1 => "one");
    }
}
```

When compiled by Phalanger, the generated class contains the following members (method bodies are omitted).

⁴ For fine-grained control, the `Export` attribute can also be applied to individual members. On the other hand, it can be also defined on the entire compilation unit (assembly) making all contained types exported.

```

[Serializable, ImplementsType]
public class A : PhpObject
{
    // Constructors
(C1) static A();
(C2) protected A(SerializationInfo, StreamingContext);
(C3) public A(ScriptContext context, DTypeDesc caller);
(C4) public A(ScriptContext context, bool newInstance);

    // Methods
(M1) public virtual object __construct(ScriptContext,
    [Optional, DTypeSpec(3, 0x2000004)] object t);
(M2) public static object __construct(object instance, PhpStack stack);
(M3) private static object Foo(ScriptContext context);
(M4) private static object Foo(object instance, PhpStack stack);

(M5) /* privscope */ static object <^Getter>(object instance);
(M6) /* privscope */ static object <^Getter>(object instance);
(M7) /* privscope */ static void <^Setter>(object instance, object value);
(M8) /* privscope */ static void <^Setter>(object instance, object value);
(M9) /* privscope */ object <Mediator>(ScriptContext, object);

(M10) public static void __InitializeStaticFields(ScriptContext context);
(M11) private static void __PopulateTypeDesc(PhpTypeDesc);
(M12) private void <InitializeInstanceFields>(ScriptContext);

    // Fields
    [PhpHasInitValue]
(F1) public PhpReference a;
    [ThreadStatic, PhpHasInitValue]
(F2) protected static PhpReference b;
(F3) public static readonly object c;

    [ThreadStatic]
(F4) private static ScriptContext <lastScriptContext>;
(F5) public static readonly PhpTypeDesc <typeDesc>;
}

```

Attributes

Classes produced by Phalanger are always serializable, because every object in PHP can be serialized. In addition, all types are decorated by the `ImplementsType` attribute in order to be recognized by the Phalanger compiler. Types without this attribute are treated as ordinary CLR types.

Base type

Classes are rooted in the abstract `PhpObject` type, which is the third and last descendant of `DObject` that we've seen. There is no need to split PHP objects into a wrapper part and a real object part, so all classes derived from `PhpObject` act as their own real objects, providing both the runtime infrastructure and the implementation.

Constructors

Static constructor (C1) contains initialization of class constants that are represented by readonly static fields - (F3) in our example. In general, it is impossible to represent class constants by literal fields since they can be initialized by other (non-class) constants, not necessarily known at compile

time. Static constructor can also initialize application-static fields⁵, which this class does not declare. Yet another task performed by the static constructor is initialization of the (F5) static field (see below).

The protected deserializing constructor (C2) contains an upcall to the base constructor with the same signature. PHP objects implement `ISerializable` and also make use of further serialization-related interfaces like `IDeserializationCallback` and `IObjectReference` to implement serialization whose behavior is compatible with that of PHP.

The two remaining instance constructors (C3) and (C4) are used when instantiating the class from PHP code. (C3) contains a logic that invokes a suitable constructor method based on the provided caller descriptor (i.e. the context in which the instantiation is being performed), whereas (C4) does not call the constructor method. The latter is used in situations when the instantiating code is able to determine the constructor method at compile time (optimization) or when e.g. cloning or deserializing an object.

Methods

(M1) and (M3) are the so-called argful overloads of the declared PHP methods. The signatures correspond to formal parameters, their type hints, and the fact whether particular parameters are optional or mandatory. These overloads contain the actual implementation and are directly called when the invocation is not „too dynamic“ in certain aspects. On the other hand, (M2) and (M4) having the unified (`object`, `PhpStack`) signature are called argless overloads. Delegates to these methods are stored in internal tables and used when the corresponding argful could not be called because it was unknown at compile time or for other reasons.

(M5) – (M8) are getters and setters of the two fields (F1) and (F2). Delegates to these methods are stored in internal tables in order to provide fast indirect access to the fields. Their purpose is identical to the getters and setters of CLR types described in the previous chapter. However, as we are now able to do so, we can generate the accessors to the resulting assembly, which saves us of from performing slower runtime generation of dynamic methods that would do the same job.

The mediator (M9) is a workaround of the, in our opinion, overly restrictive rule of the CLR PE verifier, which does not allow non-virtual calls to be made to virtual methods from static methods of the same class. The `__construct`'s argful is virtual and its argless is static. The argless contains a call to the argful but this call must be non-virtual to suppress the virtual dispatch. PE verifier reports an error if such a call exists. So instead, the argless stub calls the instance non-virtual mediator (M9), which in turn calls non-virtually the argful implementation. (M9) is therefore just a passthru stub, hopefully inlined by the JIT.

The static fields' initializer (M10) is guaranteed to be called at least once before any thread-static field access is made to the declaring class in the context of a particular web request. It contains initialization of thread-static field (F2) and is equipped with a mechanism that prevents repeated initialization (see (F4) below). Our IL emitter uses simple control flow analysis to determine when it is

⁵ Static fields are by default thread-static in PHP, but can be decorated by the `AppStatic` attribute in Phalanger to turn them into true static (so called application-static) fields.

necessary to emit a call to (M10). The name of this method is relatively sane, since we need to be able to declare it in C# in certain system classes included in our Core or Base class library.

The dynamic type descriptor populator (M11) is a runtime replacement for the slow reflection. When the Phalanger runtime needs to “reflect” a PHP type, it suffices to find this method and invoke it. (M11) populates the provided type descriptor with delegates to all those setters, getters, and argless stubs. Nevertheless, when a class is referenced by another code that is being compiled, a full reflection is triggered in order to obtain appropriate `FieldInfos` and `MethodInfos`. So basically, (M11) is only a runtime thing and the reason for having a sane name is the same as for (M10).

Finally, (M12) initializes the instance field (F1). This initialization code is pulled out to a method to avoid duplication of code since it might be needed at multiple places.

Fields

(F1) and (F2) implement the two declared PHP fields. Note that they are both decorated by the `PhpHasInitValue` attribute, because the compiler needs this information when compiling a potential derived class in order to adhere to all PHP inheritance rules. (F3) implements the class constant.

(F4) holds a reference to the last context seen by the static field initializer (M10). If (M10) is invoked in the context that is identical with the one stored in (F4), it does nothing because thread-static fields have already been initialized in this context. Otherwise, it initializes the fields and stores the context to (F4). The Phalanger runtime uses the notion of type descriptors when working with types. A type descriptor can be regarded as a dynamic add-on to `System.Type`. Every type generated by Phalanger contains a static field (F5), which is initialized in static constructor and holds a reference to the type desc of the type.

Anatomy of an Exported Type

If the same PHP class that we’ve dissected in the previous part is decorated by the `Export` attribute, the generated CLR class will look as follows.

```
[Serializable, ImplementsType]
public class A : PhpObject
{
    // Constructors
    public A();
    public A(A t);

    // Methods
    public virtual object __construct();
    public virtual object __construct(A t);
    private static object Foo();

    // Properties
    public virtual object a { get; set; }
    protected static object b { get; set; }
    public static object c { get; }

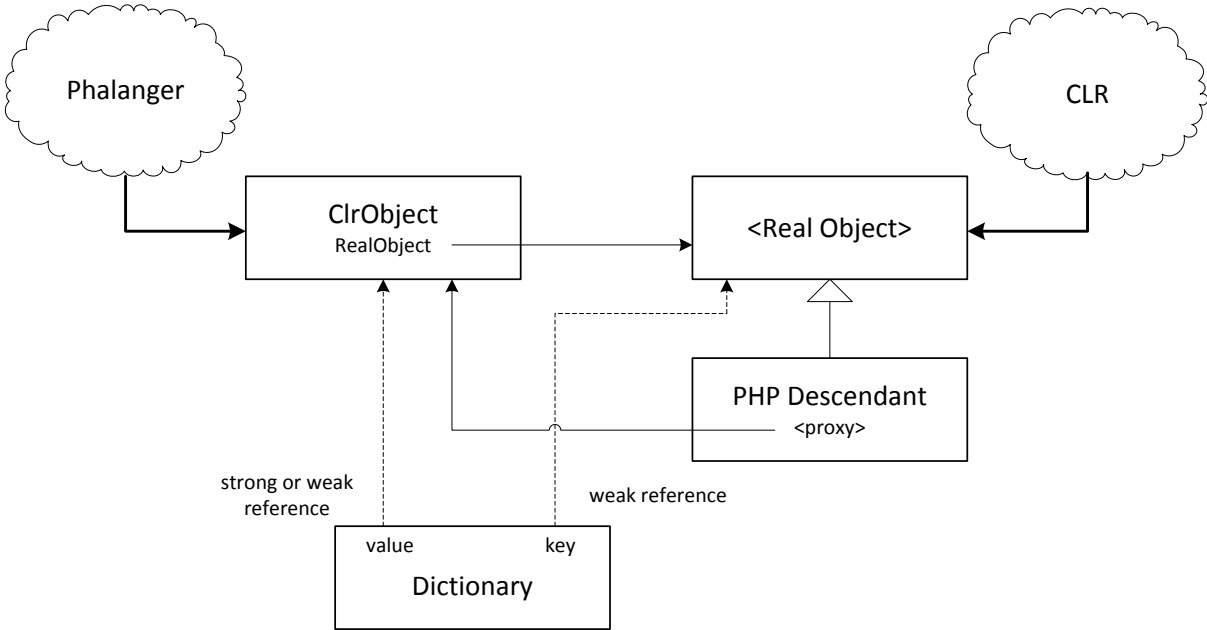
    // The same members as before (only fields are renamed
    // to avoid clashes with the properties).
}
```

Constructors are generated according to the constructor method effective for the respective class (`__construct` in this case). If there is no constructor method, the default public parameterless constructor would be automatically generated. As the constructor method is declared with one optional formal parameter, two overloads are generated – one with the parameter and one without. Parameter type hints are used to determine parameter types (`System.Object` by default). Read-write properties are generated for PHP fields and read-only properties for PHP class constants. Types of the properties are currently always `System.Object`.

Current version of Phalanger lacks the possibility to adjust types of the properties that export PHP fields and constants. Likewise, method return types are also not user adjustable. Methods return `System.Object` unless their formal parameters are recognized as standard .NET event handler parameters⁶, in which case the return type is `void`. This should be improved in future versions.

Extending CLR Types

By extending CLR types, we mean being able to declare a class in PHP that implements one or more CLR interfaces and/or derives from a CLR class. Deriving from a user-specified base class instead of the default internal `PhpObject` means that our object no longer fits into the `DObject` hierarchy and the wrapper – real object decoupling is necessary. However, as now in contrast to the simple consuming case we have some control over the real object, we can do it in a more efficient way.



The first PHP descendant of the real object adds a field named `<proxy>` that contains a reference back to the wrapper associated with this real object. The pair thus becomes mutually interlinked and

⁶ Exactly two parameters; the first is of `System.Object` type, the second is assignable to `System.EventArgs`.

if at least one of them is referenced by a strong reference, no finalization can occur. The dictionary may even be replaced by an internal interface secretly implemented by the PHP descendant that would use the `<proxy>` field to return the wrapper associated with the real object (currently not implemented). Nevertheless, the code emitted to the PHP descendant makes use of the `<proxy>` field in many situations. Essentially, the `$this` reference in PHP classes that derive from CLR classes is reachable using `ldarg.0; ldfld <proxy>` instead of the simple `ldarg.0` in classes derived from `PhpObject`.

Overriding Methods

Methods are implemented and overridden by name in a case-insensitive way. It means that by declaring a PHP method `foo`, all inherited non-final virtual methods (coming from the base type as well as from interfaces) named `foo`, `Foo`, `FOO` etc. will be overridden/implemented by this method regardless of their signature. Stubs with appropriate signatures are automatically generated to the declaring class in order to adapt the method to the required signatures. Sometimes it is even necessary to generate such a stub to a class that does not declare the target method. We call it a ghost stub and it typically arises from a code like this.

```
class A
{
    function GetEnumerator()
    {
        // ...
    }
}

class B extends A implements System::Collections::IEnumerable
{
}
```

Class `B` implements an interface and inherits its implementation. A stub that adapts `A::GetEnumerator` to `IEnumerable::GetEnumerator` has to be generated to `B`.

Overriding Properties

Properties are implemented and overridden by name (case-sensitive) by PHP fields. The property is automatically backed by the field so there is no need to explicitly specify accessors. In fact, accessors cannot be currently specified at all⁷. However, the following little trick can be employed if there is a need to execute a code upon property reading or writing.

⁷ The adjustment of the PHP grammar has been proposed and will be implemented in some of the future versions.

```

class Handler implements System::Web::IHttpHandler
{
    public $IsReusable;

    function ProcessRequest($context)
    {
        // ...
    }

    function __construct()
    {
        unset($this->IsReusable);
    }

    function __get($name)
    {
        if ($name == "IsReusable")
        {
            // getter code goes here
        }
    }
}

```

The `IHttpHandler` interface contains a read-only property named `IsReusable`. We have to add a field of the same name to indicate that we implement that property. When instantiating the class, we unset the field, which in PHP effectively means that we erase it from the instance. In addition, we add the `__get` magic method that is invoked whenever an unknown field is read. Since `IsReusable` is now an unknown field, we have full control over the property getter. A similar trick works for setters, too.

Invoking Base Constructors

There is a significant difference in constructor semantics between CLR and PHP. Constructor methods in PHP behave as ordinary instance methods, only with the slight detail that they are invoked after a new instance of the class has been created. On the other hand, CLR constructors are not explicitly callable and not inherited. Moreover, every CLR constructor has to unconditionally invoke one of its base type's constructors in order to be verifiable.

When a PHP class extends a CLR class with a non-trivial constructor, i.e. with a non-zero number of formal parameters, we have to enforce an explicit upcall from within the PHP constructor method. Since this upcall must in fact be moved from the constructor method to the real constructor to satisfy CLR rules, we have decided to introduce a new syntax for doing so. If there is such non-trivial constructor in the base class, the PHP class must declare a constructor method and indicate arguments for the upcall using the `: parent()` syntax as illustrated in the following snippet.

```

class MyException extends System::Exception
{
    function __construct($message) : parent($message)
    {
        // ...
    }
}

```

References

- [1] The Phalanger project hosted on CodePlex
<http://www.codeplex.com/Wiki/View.aspx?ProjectName=Phalanger>
- [2] Microsoft .NET Framework
<http://msdn.microsoft.com/netframework/>
- [3] The PHP interpreter
<http://php.net/>