P3 Class Definitions

Peter Vanderbilt
pvanderbilt@acm.org

Version 3.2.3

Abstract

This technical note describes a subset of P3 class definitions, defining their syntax and providing an informal semantics. A P3 class can have complex content and virtual static fields. A class definition yields two instance types, exact and general, and a class object. The subset of class definitions described in this note corresponds to the domain of the class encoding described by technical note Encoding and Typing of Complex Class Definitions [14].

This technical note is one of a collection that describes the research of the author; see [15] for more details.¹

1 Introduction

In object-oriented programming, a class is a construct used to create objects. Classes, through inheritance, provide reusable code. In P3, a class definition, when executed, causes a class to come into existence and to be bound to a given name. This technical note describes P3’s class definitions.

The classes of P3 are modeled on those of C++ [9], Java [5] and Scala [8]. However, P3 classes can have complex content and virtual static fields (of complex content). Also P3 classes are (partially) first-class objects.

1.1 Complex content

An object with complex content is one that, in addition to regular data fields, may have type-valued fields, class-valued fields and fields containing objects with complex content. We call such objects hybrid objects, or just hybrids. If a hybrid, h, has a type alias for T, the expression “h.T” references h’s alias for T. The instances of a class with complex content are hybrids.

The special variable self denotes the future instance object. Since it is a hybrid, self can contain type aliases and, so, self may appear in the class’s member types (as well as in member implementations).

¹Copyright © 2014-2019 by Peter Vanderbilt. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit creativecommons.org/licenses/by-nc-nd/4.0/.
1.2 Two instance types per class

In P3, a class with virtual complex content results in two instance types as a way to handle certain problematic situations. For class C, we call these the exact type, C@, and the general type, C#.

- Type C@ is the type of instances of just class C; it captures the actual values of the type variables and thus supports full access to an object (including the “problematic” method calls).

- Type C# is the type of instances of C and its subclasses; it only constrains the type variables, doing so in a way to support subclass polymorphism. However, problematic method calls fail typing (as desired) for objects of type C#.

The following relationships are true of class C and its superclass D:

\[
\text{new } C : C@ <: C# <: D# \quad (1.1)
\]

However, in general, C@ and D@ (if D is concrete) are not compatible.

1.3 Class objects

A class object is a first-class representation of a class: it is an object generated from a class’s definition that contains some or all of the class’s functionality. For the purposes of this technical note, a class object contains

- a field containing the type of the class’s instances,

- a function for creating such instances, and

- any static fields declared in the class or its base classes.

A class object is a hybrid since it contains a type field (the first item) and potentially contains complex content in the static fields. Class objects will be described further in section 3.2.

A class object is first-class in the sense that it can be passed to functions and stored in data structures. Class object types classify class objects in useful ways. In particular, each class defines a type that contains itself (if concrete) and all concrete subclasses (direct or not).

Within the body of a class definition, the future class object is denoted by Self (note the capital “S”). Because the class object has complex content, Self may appear in the class’s types (both member and static).

1.4 Virtual static fields

Field definitions are either member or static: Member fields appear in instances created by “new C”, while static fields appear in the class object, C. As previously mentioned, within a class definition, self refers to the future instance and Self refers to the future class object.
Because the class object contains static fields, it is possible for those fields to be virtual, in the sense that they inherit and can be overridden, with the latest-bound implementation used. We choose to give static fields all the modes that member fields have, including virtual and abstract. Thus static fields are late-bound with Self in the same way that member fields are late-bound with self.

## 2 Syntax of class definitions

The following grammar (see *P3 Preliminaries* [13]) defines a syntax of P3 class definitions:

\[
\begin{align*}
J^C & \ ::= \ Q^c \ \text{class} \ C \ (\bar{p}:\bar{T}^p) \ \text{extends} \ D(\bar{a}) \ \{ \bar{F} \} . \\
Q^c & \ ::= \ \text{open} \ | \ \text{partial} . \\
F & \ ::= \ ( \text{member} | \text{static} ) J .
\end{align*}
\]

Non-terminal \(J^C\) is one alternative in the grammar of P3 definitions. Non-terminals \(C\) and \(D\) range over identifiers, \(\bar{p}\) over identifier lists, \(\bar{T}^p\) over type lists, \(\bar{a}\) over expression lists and and \(\bar{F}\) over field descriptor definition lists (to be described in section 2.1). Lists \(\bar{p}\) and \(\bar{T}^p\) are parallel, but \(\bar{p}\), \(\bar{a}\) and \(\bar{F}\) are independent. Certain field names are reserved and, so, can not be defined by the programmer; “Instance” and “Class” are two such names.

In convenient form, the keywords open and member may be omitted, “C()” may be written just “C,” “D()” may be written just “D” and “extends Object” may be omitted.

Identifier \(C\) is a name to be bound to the newly defined class. The phrase “(\(\bar{p}:\bar{T}^p\))” describes the constructor parameters (the arguments of “new \(C(...)\)”). Identifier \(D\) names a superclass, a class from which fields are inherited; it is Object by default. The length of arguments \(\bar{a}\) must conform to \(D\)’s constructor parameter specification. Field definitions \(\bar{F}\) form the body of this class, each of which defines a member or static field. The member keyword indicates that the definition relates to the instance, while static indicates that the definition relates to the class object (see 3.2). As discussed in 1.4, static fields have the same modes as member fields, including abstract and virtual. Each \(J\) is a field descriptor definition, which will be described further in section 2.1.

### 2.1 Syntax of field descriptor definitions

The syntax of field descriptor definitions is given in section 2 of *P3 Field Descriptor Definitions* [12]. In short, there are two kinds of field descriptors, virtual and abstract. A virtual field descriptor has the following components:

- a name,

- an implementation, which is an expression that evaluates to the field’s initial (or only) value,

- a “current” type, which is the type of the implementation, and

- a “future” type, which is a type that constrains future values.
An abstract field has only a name and a future type. For data fields, the current and future types are the same, but they differ for complex content.

The current value of a field is the one appearing in an instance created by “new C”. A future value of a field is one appearing in some (possibly indirect) extension of C (including C).

Field descriptors are also inherited from the superclass, D. We call these pre-existing field descriptors. There are field descriptor definitions that (in class C):

- create new field descriptors,
- provide an implementation of a pre-existing abstract field, yielding a virtual one,
- override the implementation of a pre-existing virtual field, or
- refine the future type of a pre-existing field.

Pre-existing field descriptors (from D) that have no matching definition is C are inherited from D into C.

2.2 Concreteness qualification

We define that a class is concrete if it is complete enough that instances can be constructed. In contrast a class is partial if it is not complete enough for implementation, typically because one or more field implementations are missing. A concrete class must not have abstract fields (either member or static, either directly defined or inherited), while a partial class may.

We define that a class is extensible if it is designed to support inheritance. This generally means that implementations are polymorphic over future classes. In contrast, a class is closed if it is not designed for extension.

Non-terminal Qc is the concreteness qualification:

- open indicates that the class is concrete and extensible.
- partial indicates that the class is partial but extensible. A partial class does not generate instances, nor does it have a class object.

2.3 Special variables

Within a class definition, the special variables self and Self refer to the future instance and future class object, respectively. In particular, if an object created by “new C” is assigned to path p, then references to self are bound to p and Self to C. As is traditional, a sibling field with name “x” can be referenced directly, as just “x”, but the canonical form is “self.x” (or “Self.x” if static).

Additionally, the special variables super and Super refer to records containing member and static fields (respectively) inherited from superclass D.

The type Self@ is the type of instances of the future class denoted by Self and is referred to as the self type [6, 3, 10, 1, 7] and corresponds to Bruce’s MyType [3, 4, 2].
The variable self enables mutual recursion among member fields, including type fields and other complex fields. In particular, self can support extensible mutual recursion between member type fields. Similarly, variable Self enables extensible mutual recursion between static type fields.

3 Result of class definitions

A class definition results in several entities, including various types, a class object and mechanisms for extension. A rigorous definition of these entities is given by our encodings, such as Encoding and Typing of Complex Class Definitions [14].

Here we present an informal, encoding-neutral description of the entities created by a class definition. In particular, the open-form class definition of (2.1) creates at least the following entities:

- An implementation of “new C(¯e)”, the expression that creates instance objects of class C where the values of ¯e are bound to ¯p (of (2.1)). The expression “new C()” may be written “new C”. Instance objects will be described in section 3.1. As described in section 3.2, “new C” is actually an aspect of the class object C. “new C” is formed from the implementations of the member field descriptors.

- Two instance types: an exact type C@ and a general type C#, as discussed in section 1.2 and satisfying assertion (1.1). As will be described in section 3.2, type C@ is actually an aspect of the class object C. Type C@ is formed from the current types of the member field descriptors, while C# is (indirectly) formed from their future types.

- A class object C, to be described in section 3.2.

- A class object type { {?< <C} } whose instances are the class object of C and those of C’s subclasses. It will be described further in section 3.3.

- An implementation of “extends C” (which supports inheritance from C).

A class definition marked partial does not generate “new C”, C@ nor class object C.

3.1 Structure of instance objects

The expression “new C(...)” creates instances of C. These instances contain those member fields declared in C or inherited from D (including those from D’s super-classes). Member fields are accessed as o.m for instance object o and member field name m.

3.2 Structure of class objects

The class object is a first-class object named C that contains at least the following fields:

- Those fields declared static in C or its superclasses. Static fields are accessed directly, as C.s for static field name s. C’s static fields come from the implementations of the static field descriptors.
• A type field named `Instance` that is the type of instances created by class `C`. We define type `C®` to be syntactic sugar for the `Instance` type field:

\[ C® \triangleq C.\text{Instance} \quad (3.1) \]

• A function `new` that creates instances of class `C`. The following syntactic sugar is used to access `new`:

\[ \text{new } C(\bar{e}) \triangleq C.\text{new}(\langle\bar{e}\rangle) \quad (3.2) \]

For technical reasons, the parameters, `\bar{e}`, are packaged as a tuple and passed to `C.new`. “`new C`” is an abbreviation for “`new C()`”.

### 3.3 The extends relation

Class type `{ {?<<C}}` has as its instances the class object of `C` (if concrete) and those of `C`’s concrete subclasses. It is a hybrid type with declarations for `C`’s static fields and type `Instance`. It is formed using the future types of the static fields with `Instance` constrained to subtype a type formed from the future types of the member fields.

We define the following syntactic sugar:

\[ B << C \triangleq B : \{ {?<<C}} \quad (3.3) \]

Thus `<<` can be viewed as a relation between classes such that `B<<C` means that `B` is a concrete subclass of `C`. For a variable `B` bounded only by extends (`B<<C`), `B®` is defined (as an alias for `B.\text{Instance}` by (3.1)), but none of `B#`, `{ ?<<B}` nor “`new B`” are defined; “`new B`” is undefined because `B`’s initialization parameters are not known.

### History by version number

3.2.3 (Apr 2014) Fixed PDF URLs (for new website).

3.2.2 (Feb 2014) Added copyright and Creative Commons license.

3.2.1a (Jul 2013) Fixed cross-references.

3.2 (Jul 2013) Added introductory text with subsections, added partial, improved the grammar and description of class field descriptor definitions. Moved field descriptor text to P3 Field Descriptor Definitions [12].

3.1 (Apr 2013) Extracted from my Reduction and typing of class definitions with complex content paper.
References


