Object Creation in Grace

MICHAEL HOMER and JAMES NOBLE, Victoria University of Wellington

We are engaged in the design of Grace, a new object-oriented open source programming language aimed at instructors and students in introductory programming courses. Grace aims to include features that have been found useful in software practice, while allowing multiple different teaching approaches without requiring that concepts be introduced to students before they are ready. While many aspects of Grace's design will be familiar to most object-oriented programmers, Grace cleanly separates the concepts of "object", "class", and "type", and so Grace offers more options for creating objects than most other languages. We have written these patterns to explain how Grace programmers should go about creating objects.

Categories and Subject Descriptors: K.3.2 [**Computers and Education**] Computer and Information Science Education

Additional Key Words and Phrases: Object creation, programming languages, Grace

ACM Reference Format:

Homer, M. and Noble, J. 2015. Object Creation in Grace. EuroPLoP '13: Proceedings of the 18th European Conference on Pattern Languages of Program, Article 21 (July 2013), 15 pages. ACM.

1. INTRODUCTION

We are engaged in the design of Grace, a new object-oriented open source programming language aimed at instructors and students in introductory programming courses [Grace 2013; Black et al. 2012; Noble et al. 2013; Black et al. 2013]. Grace aims to include features that have been found useful in software practice, while allowing multiple different teaching approaches without requiring that concepts be introduced to students before they are ready. Minigrace, a self-hosted prototype implementation of the language that compiles either to C or JavaScript is available from the Grace project website (gracelang.org), and includes a simple web-based interface.

While many aspects of Grace's design will be familiar to most object-oriented programmers — we hope Grace's syntax, declarations, control structures, and method semantics appear relatively conventional — there are some aspects of Grace's design that are unique. In particular, Grace tries to separate the concepts of "object", "class", and "type". In Grace, objects can be created without classes (like JavaScript or Self), or with classes (like most other object-oriented languages). Classes in turn can be built "by hand" out of objects (like Emerald, if anyone's counting). Grace types are optional (pluggable): types in Grace programs may be left out all together (like Python or JavaScript), always included (like Java or C♯), or used in some mixture (like Dart or Dylan).

The separation of object, class, and type does not really affect most Grace programming: variables and methods are used in Grace as they are in most other languages. Where this separation affects programming is when objects are created: Grace offers more options for creating objects than most other languages, and the "division of labour" between Grace's object creation constructs is different to most other languages. Thus, we have written these patterns to explain how Grace programmers should go about creating objects.

DOI: http://dx.doi.org/10.1145/2739011.2739032

We can imagine four audiences for these patterns. First, there are potential Grace programmers, experienced programmers who are considering learning Grace or even contributing to the Grace project in some way. Second, there are potential teachers, lecturers, tutors, or professors who are considering adopting Grace, or who are expecting to teach Grace. We expect both potential programmers and potential teachers to have somewhat similar backgrounds: to have experience with at least one other object-oriented language, typically Java, C♯, Python, or JavaScript, perhaps Scala or Objective-C; and to have some awareness of the "Gang of Four" design patterns, especially the creational patterns Singleton, Abstract Factory, and Factory Method. The next potential audience is ourselves, the rest of the Grace design team, contributors, and other language designers. Grace is still a work in progress: each of these patterns embodies a scenario or use-case for Grace's object model. We hope to use feedback about these patterns not only to improve our explanations of Grace's design, but also to improve that design itself. Finally, the last potential audience for these patterns are students learning to program. Perhaps a few of them will find these patterns useful, but, realistically, most novices need many things expressed in fine detail before they can understand most of the design issues addressed by these patterns. We hope that in time there will be better resources for novices to learn Grace (and to learn to program in Grace), and that these patterns may be useful resources for instructors developing those teaching materials.

The next section gives a high-level overview of Grace, with particular attention to features used in these patterns. The following sections describe patterns for simple object creation, classes, and singleton objects. As Grace is a new language still in design, there are few known uses of any of these patterns, although many are used in the Minigrace compiler for the language, and the libraries we are developing for the language.

2. GRACE

Grace is a pure object-oriented language with many familiar features: Grace is memory-safe, objects are passed by reference, garbage-collected, uses syntax derived from Java and C, and it is open source. Grace has been designed to look familiar to users of other languages but to allow multiple teaching sequences and styles, and its features have been chosen with that in mind.

Mutable and immutable bindings are distinguished at the keyword level: **var** defines a name with a variable binding, which is set and changed using the ":=" operator, while **def** defines a constant binding initialised using "=". The same keywords are used to define local variables and object fields, and define accessor methods in objects.

A Grace method name may contain multiple "words". After each word is an argument list, allowing parameters to be labelled with their role:

```
method test(x)between(y)and(z) {
  if (x < y) then {
    return false
  }
  if (x > z) then {
    return false
  }
  true
}
test 5 between 3 and 7 // -> true
```

Grace is gradually typed. Code may be written with or without types, or with a mixture of the two, allowing instructors to choose when to introduce types, and to do so in the same language as used for untyped code. When typed and untyped code are combined, runtime type-checks occur when values pass from dynamic to static code.

All Grace types are structural: the type of an object is determined solely by the methods it contains, and not by explicit reference to any named types. New types may be created after the fact and existing objects may conform to them. Types may be named with the **type** keyword or may be anonymous:

```
type Point = {
  x -> Number
  y -> Number
}
// Anonymous type { x -> Number }
method height(p : type { x -> Number }) { p.x }
```

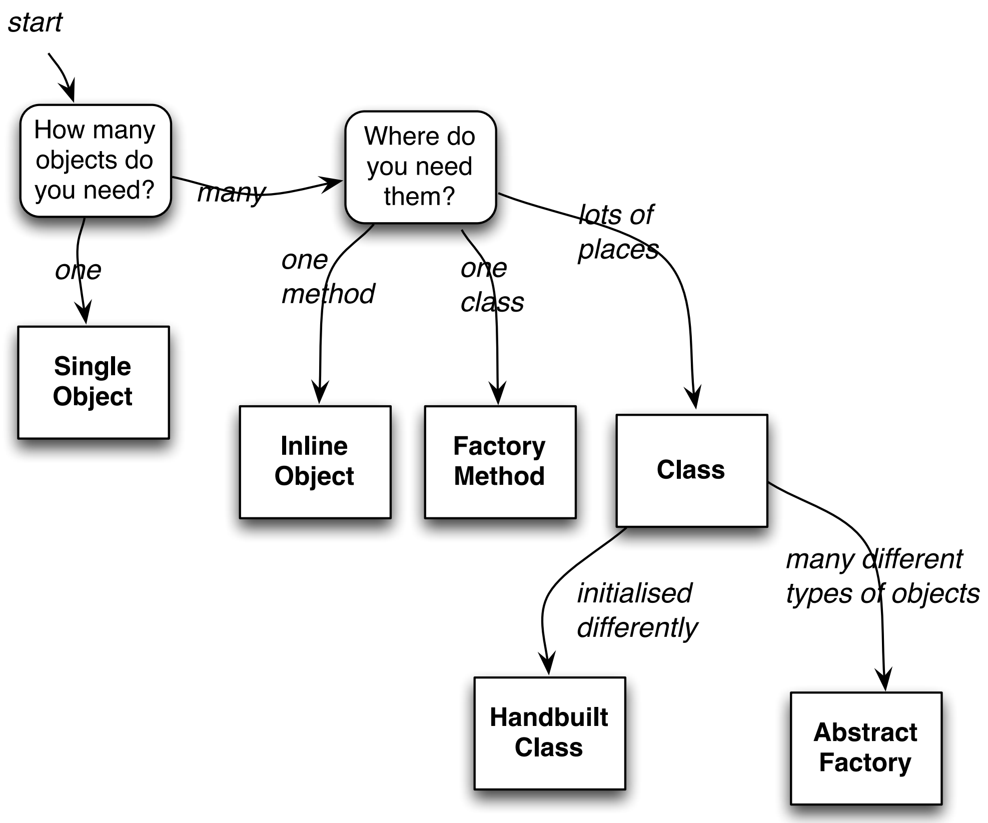
Type operators allow specifying a type that is a combination of other types. Crucially, types and implementation are strictly separated from one another. Many objects may belong to a type without having anything else in common with one another.

3. PATTERNS

This paper presents six key creational patterns in Grace, in the order they would be taught, from simple to more complex.

Single Object	How can you create a single object? Use an object constructor and a **def** to give it a name
Inline Object	How do you create many similar objects? Use an object constructor inline in code
Factory Method	How do you create many similar objects with the same methods but different state? Use an object constructor inside a Factory method
Class	How do you create many similar objects from different parts of your program? Use a class declaration.
Hand-built Class	How do you provide multiple ways of making an object? Use a hand-built class definition.
Abstract Factory	How do you create many different kinds of objects in the same way? Use a type definition to represent an abstract factory.

These six patterns aim to cover the core techniques for creating objects, in order of increasing complexity The first pattern, Single Object, is the most straightforward in practice: running an object constructor once and binding the result to a name. The second pattern, Inline Object, generates many similar objects by running an object constructor inline in a method body. The next two patterns, Factory Method and Class, factor out object creation using the two abstraction mechanisms in Grace: methods and objects. The final two patterns are more advanced, describing how to offer different interfaces to create objects, and how to abstract over those interfaces.



4. FORCES

As with any design decision, all the patterns in this paper make tradeoffs between important design considerations — the *forces* that underly the design problems to be solved. Patterns typically resolve one or more forces, so they no longer need to be considered in that part of the design, but solving one problem can often give rise to new forces that must be resolved in turn.

The main forces these patterns address include:

—*How many objects do you need to create?* Most of these patterns can create multiple objects, although some patterns create just one object.

—*How different are the objects you need to create?* Some of these patterns create objects that a very similar to each other, others construct objects with different field values but similar structures, while other patterns can create objects with different structures.

—*How much of the program needs to create these kinds of objects?* Some kinds of objects are very special purpose, and are only ever created in one particular place in the program. Other kinds of objects, perhaps more generally useful, may need to be created in many different places throughout the code.

—*Modularity* How cohesive is your code? How tightly coupled to other parts of the program? How far does a change to one part of the program ripple through to other parts of the program?

—*Simplicity* Ideally, we want "The Simplest Thing That Could Possibly Work" [Cunningham 2005]. Unfortunately, we don't live in an ideal world.

5. SINGLE OBJECT

How can you create a single object? Sometimes you need just one solitary object used in only one place.

Forces:

—You want to create exactly one object.

—You want to ensure that all code accesses the same object.

—You may wish to create multiple instances in the future — but not yet.

—You want to avoid overhead.

Therefore: *Use an object constructor, and a* **def** *to give it a name.* An object constructor creates a fresh object. A **def** declaration creates a constant, immutable name binding. Assigning an object constructor as the value of a **def** creates a singleton object accessible by that name. If you wish to allow creating multiple instances in the future, you can replace the declaration with a method without changing the interface.

Example

Suppose we have a plotting system that plots points on a display. Points should know their own x and y coordinates, and be able to calculate their distance from the origin. First, we'll need to create an object that represents the origin of the system. We can create this origin object with an object constructor, and then use a **def** to give it a name:

```
def origin = object {
  def x is public = 0
  def y is public = 0
  method distanceFromOrigin { return 0 }
}
```

Grace object constructors construct a new object when they are executed. Object constructors use the **object** keyword and may declare methods and fields of the new object. This object constructor declares an object with x and y fields that can be read from outside the object (thanks to is public) and a method that returns zero distance from the origin (methods are public by default).

Consequences

—**Benefits.** The code creates just one object. All references to the name will refer to the same object, which is statically determined. The code doesn't have to make a class or copy a prototype.

—**Liabilities.** Initialising the singleton may be expensive, There may be other objects of the same structural type with unconnected definitions elsewhere in the program, or even identical objects declared elsewhere.

Other Languages

Scala's object declarations [Odersky et al. 2011] create a single named object, as do object literals in Self [Ungar and Smith 1991].

```
object origin {   // Scala
    val x = 0
    val y = 0
}
```

Related Patterns

If you need to make more than one object, consider Inline Object.

6. INLINE OBJECT

How do you create many similar objects? Sometimes you need to create many similar objects, with the same structure (collections of methods and fields) but different field values. The field values to be held in the object are present in the surrounding environment or can be calculated at when you create the object.

Forces

—You need to create many different objects.

—All the objects have the same structure of specific methods and fields.

—You can calculate initial values for fields.

Therefore: *Use an object constructor inline in code* Grace's object constructors construct a new object every time they are executed. First the object's structure is built, then any code inside the constructor is executed top to bottom — including expressions that initialise **def**s and **var**s.

Example

We have been given a data table — a list of objects containing manufacturing costs for various size widgets. We want to define some point objects to make up a scatter plot of the data. We can use an object constructor to create each point in a for loop to loop over the data table.

```
def dataTable := ...

for (dataTable) do { each ->
  plot (
    object {
      def x = each.size * 5
      def y = each.cost * 10
      method distanceFromOrigin { return (x^2+y^2).sqrt }
    }
  )
}
```

Here we create the point objects inline, and pass them directly to the plot method to draw each point. We can make method requests on an inline object, store a reference to it in a variable, or pass it to other code — just like any other kind of object. Note how the inline object has obtained some of its initial state from the surrounding lexical scope it was defined in — here, some of the attributes of the objects in the table.

Consequences

—**Benefits.** An object may be defined in-line at the point it is used. The object conforms exactly to the needs of the program at that point.

—**Liabilities.** An object constructor creates a single object when it executes, and can only depend on data from the surrounding lexical scope. If many similar objects are to be created separately they must all be updated when any changes are required.

Other Languages

JavaScript's object literals [Crockford 2008] create an object each time they are executed, as do object literals in Emerald [Ungar and Smith 1991].

```
{       // JavaScript
  x : 0
  y : 0
}
```

Related Patterns

The Factory Method and Class patterns provide other ways of creating and initialising multiple similar objects. The Single Object pattern creates just one object.

7. FACTORY METHOD

How do you create many similar objects, with the same methods but different state? Suppose that rather than creating a single object we want to create multiple objects. The objects will have the same structure but will initialised differently.

Forces

—You need to create many similar objects.

—You want to centralise the definition so that only one location needs to be updated when requirements change.

Therefore: *Use an object constructor inside a factory method.* Whenever an object constructor executes it creates a new object, but can only initialise that object from its surrounding lexical scope. Methods can have parameters, so by putting an object constructor into a method, we can abstract over the values to be initialised. A method returning an object constructor may use the method arguments to initialise the object it creates.

Example

```
method createPoint(x' : Number, y' : Number) −> Point {
  object {
    def x = x'
    def y = y'
    method distanceFromOrigin {
      (x^2+y^2).sqrt
    }
  }
}
def myPoint = createPoint(1, 2)
```

We can call createPoint many times with different arguments, getting back a new point each time, with whatever coordinates we want. The annotation "−> Point" says that the object returned must have the type "Point"

Consequences

—**Benefits.** Many objects, differing only in the initialisation of some of their internal state, can be created from different points in the code. The required state values are explicit. If the shape of the objects requires updating, only one source location must be changed.

—**Liabilities.** The factory method must be defined before use. Other code creating these objects must be able to invoke the factory method. The behaviour of a factory method (that it creates a new object) is not as clear as an Inline Object.

Related Patterns

The Class pattern creates an object that can be used for creating other objects, putting the method running the object constructor into a separate object dedicated to that purpose. The Hand-built Class pattern and Abstract Factory patterns also use factory methods. Natural Creation makes new objects by adding Factory Methods to existing objects in the domain model [Noble 1999].

8. CLASS

Also Known As: Factory Object

How do you create many similar objects from different parts of your program? Suppose that we want to create many similar objects, initialised differently, from different objects or different modules. We need to create these objects, without client code needing to care about the internal structure of the objects being created, and without client code having to be contained in the same object or module as a factory method or inline object constructor.

Forces

—You need to create many similar objects.

—These objects can all be initialised in the same way.

—You wish to centralise definitions so that all creation of your objects happens through a single object.

Therefore: *Use a class declaration.* A class declaration creates a class: an object that creates other objects via a Factory Method. A class declaration also declares a name to refer to the class.

Example

We can write a class declaration to make many points:

```
class point.x(x')y(y') −> Point {
  def x = x'
  def y = y'
  method distanceFromOrigin { ((x^2) + (y^2)).sqrt }
}
```

and the request the class to create new objects:

```
def myPoint = point.x(fromLeft)y(fromBottom)
```

The return type annotation "−> Point" links the class "point" with the type of object it creates "Point". Types and classes are separate in Grace: we write class and method names beginning with a lowercase letter and type names beginning with an uppercase letter.

Consequences

—**Benefits.** Many objects may be created from a single definition. The class object may be provided to other code to allow that code to create this kind of object. Classes are familiar to programmers from most other object-oriented languages — hopefully making Grace easier to learn, and helping students transition from Grace to other languages.

—**Liabilities.** Only a single sort of parameterised initialisation is possible from a class definition. A class must provide for every valid initialisation from the same constructor.

Discussion

A Grace class declaration is syntactic sugar for nested object constructors. Our class is equivalent to:

```
def point = object {
  method x(x')y(y') {
    object {
      def x = x'
      def y = y'
```

```
            method distanceFromOrigin { ((x^2) + (y^2)).sqrt }
        }
    }
}
```

Related Patterns

A Class in Grace is just an object containing a Factory Method. A Hand-Built Class can support multiple factory methods.

9. HAND-BUILT CLASS

How do you provide different ways of making an object? A class declaration creates an object with one factory method, but suppose we want to have multiple ways of initialising objects the class creates?

Forces

—You want to create a number of similar objects.

—Those objects need to be initialised in different ways.

—You wish to centralise definitions so that all creation of your objects happens through a single object.

Therefore: *Use a hand-built class definition.* A class definition makes an object containing a factory method, and gives that object a name. In a hand-build class definition, you write out the full definition of that class object yourself, and are able to extend or customise its behaviour.

Example

This Hand-Built point "class" object has two methods that create new points, using either Cartesian or polar coordinates. This class also counts the number of points created.

```
def point = object {
  var numberOfPointsCreated := 0
  method x(x')y(y') {
    numberOfPointsCreated := numberOfPointsCreated + 1
    object {
      def x = x'
      def y = y'
      method distanceFromOrigin { ((x^2 + y^2)).sqrt }
    }
  }
  method r(r)θ(θ) {
    def x = r ∗ θ.cos
    def y = r ∗ θ.sin
    self.x(x)y(y)
  }
}
def myCartesianPoint = Point.x(1)y(0)
def myPolarPoint = Point.r(1)θ(0)
```

Consequences

—**Benefits.** A single class object can provide for many ways of initialising similar objects. Other code may treat the manual class exactly the same way it would treat one declared with the class keyword.

—**Liabilities.** The manual class definition is more verbose than a class declaration. If the initialisations of objects are very different, the inner object constructors may need to be repeated.

Discussion

To avoid duplication, you can implement one of the factory methods in terms of another — in the example above, the polar factory method calls the cartesian factory method. Methods and fields in Hand-Built Class objects can often take the place of the static methods or fields in languages like C++, Java, or Dart.

Related Patterns

The Class pattern uses built-in syntax to simplify many class definitions which need only one Factory Method. An Abstract Factory type is typically implemented by a Hand-Built Class.

10. ABSTRACT FACTORY

How do you create many different kinds of objects in the same way? Class definitions work well when we are creating one particular kind of object, but what if we want some code to be parameterised to create and use different kinds of object, or related families of object? In this case we would use an Abstract Factory, as in Gamma et al. [1995].

Forces

—You want to be able to make many different kinds of object with a similar purpose and interface.

—You want to be able to create families of related objects with different purposes.

—You want client code to be able to create all of these objects without caring about the implementation.

Therefore: *Use a type definition to represent an abstract factory.* Because Grace types are not coupled to implementation, we can define them to represent our Abstract Factory. As structural types, they need only contain the factory methods client code will actually be using, and many different types may exist describing the same factory objects.

Example

Suppose we are implementing the user interface for our plotting system. We want the system to work across platforms using native widgets, but do not want to clutter the body of our code with implementation details of each system. Furthermore, some platforms do not support all kinds of widget and so some functionality may be unavailable, but we want all parts of the system to work wherever they are supported.

Here we show the definitions of factories for GTK, a desktop windowing system, and for HTML, which will display in a web browser.

```
class gtkButton.new(text : String) −> Button { ... }
class gtkText.new(text : String) −> Text { ... }
class gtkScrollBar.new −> ScrollBar {
  method scroll(amt : Number) { ... }
}
class gtkWidgetFactory.new −> WidgetFactory {
  method createButton(text : String) −> Button { gtkButton.new(text) }
  method createText(text : String) −> Text { gtkTextBox.new(text) }
  method createScrollBar −> Scrollbar{ gtkScrollBar.new }
}

class htmlButton.new(text : String) −> Button { ... }
class htmlParagraph.new(text : String) −> Text { ... }

class htmlWidgetFactory.new −> WidgetFactory {
  method createButton(text : String) { htmlButton.new(text) }
  method createText(text : String) −> Text { htmlParagraph.new(text) }
}
```

HTML does not support a scrollbar widget, so the HTML widget factory does not contain the `createScrollbar` method.

To represent the generic factory interface, we can create and name a Grace type:

```
  type WidgetFactory = {
```

```
  createButton(text : String) −> Button
  createText(text : String) −> Text
}
```

which describes an object with two methods, createButton that creates a button and createText that creates a text box. Both factory classes are declared to return an object of this type: compiling a class will give an error if the objects of the class don't conform to the WidgetFactory type.

In another part of the code we can use a factory to create buttons and text boxes:

```
method makePanningArea(factory : WidgetFactory) {
  def msg = factory.createText "Hello"
  def zoomIn = factory.createButton "+"
  ...
}
```

Elsewhere, we want to create some scroll bars:

```
method makeButtons(factory : { createScrollBar −> ScrollBar }) {
  def yesButton = factory.createButton "Yes"
  def noButton = factory.createButton "No"
  ...
}
```

The makeButtons method has the minimum restriction possible on the type of factory – only requiring a createScrollbar method taking a String and returning a Button – allowing both the HTML and GTK factories to be used. The makeButtons method uses an anonymous type declaration to require only the method it will actually be using, without the overhead of declaring a type with just one method.

Consequences

—**Benefits.** Code can be parameterised over whole families of related objects. The program can guarantee that no invalid factory is provided, without restricting any further than necessary.

—**Liabilities.** Families of objects may not support similar enough interfaces without wrappers. If the type requirement of an inner method changes, the type change may need to be propagated far up the call tree. It's even less obvious that calls to Factory Methods in an Abstract Factory will create objects.

Related Patterns

The most surprising feature of an Abstract Factory in Grace is that there is actually no Abstract Factory object. An Abstract Factory itself is represented by a type, not a class; any object that conforms to that type may be used as a Factory. That said, an Abstract Factory type is typically implemented by a Hand-Built Class. Gamma et al. [1995] describe different forms of Abstract Factory.

Acknowledgements

Thanks to Stefan Sobernig, our patient shepherd for EuroPLoP2013, even if we weren't able to take nearly enough of his excellent advice, to Ernst Oberortner for PLOP 2012. Thanks also to all those who workshoped this paper at EuroPLoP — Veli-Pekka Eloranta, Carsten Hentrich, Daniel Sagenschneider, Didi Schütz, Michael Weiss, Uwe Zdun — and apologies as well as thanks to anyone we've forgotten.

REFERENCES

Black, A. P., Bruce, K. B., Homer, M., and Noble, J. 2012. Grace: the absence of (inessential) difficulty. In *Onward!* ACM.

Black, A. P., Bruce, K. B., Homer, M., Noble, J., Ruskin, A., and Yannow, R. 2013. Seeking Grace: a new object-oriented language for novices. In *ACM Technical Symposium on Computer Science Education (SIGCSE)*. 129–134.

Crockford, D. 2008. *JavaScript: the Good Parts*. O'Reilly.

Cunningham, W. 2005. Do the simplest thing that could possibly work. http://c2.com/xp/DoTheSimplestThingThatCould-PossiblyWork.html.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Grace 2013. The Grace programming language. http://gracelang.org/.

Noble, J. 1999. Natural creation: A composite pattern for creating objects. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS 32) Melbourne*. 78–88.

Noble, J., Homer, M., Bruce, K. B., and Black, A. P. 2013. Designing grace: Can an introductory programming language support the teaching of software engineering? In *26th Conference on Software Engineering Education and Training (CSEE&T)*.

Odersky, M., Spoon, L., and Venners, B. 2011. *Programming In Scala*. artima.

Ungar, D. and Smith, R. B. 1991. SELF: the Power of Simplicity. *Lisp and Symbolic Computation 4,* 3.