PEARSON Education

ADDISON-WESLEY

open source library

Andi Gutmans, Stig Sæther Bakken, Derick Rethans

PHP 5 aus erster Hand

Das Entwicklerhandbuch für Profis

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

3 Objektorientierung in PHP

»Große Gedanken benötigen eine große Sprache.« – Aristophanes

3.1 Einführung

Die Unterstützung für objektorientierte Programmierung (OOP) wurde in der Version PHP 3 eingeführt. Obwohl brauchbar, war sie sehr einfach und wurde auch in PHP 4 nicht sehr verbessert, damit die Rückwärtskompatibilität gewahrt blieb. Aufgrund des allgemeinen Bedürfnisses nach einer verbesserten OOP-Unterstützung wurde das Objektmodell für PHP 5 vollständig neu entworfen. Es gibt eine Vielzahl neuer Funktionen und Veränderungen am Verhalten des »Basisobjekts« selbst.

Wenn PHP für Sie neu ist, finden Sie in diesem Kapitel eine Einführung ins objektorientierte Modell. Auch wenn Sie mit PHP 4 vertraut sind, sollten Sie es lesen, da sich mit PHP 5 bezüglich OOP fast alles geändert hat.

Nach der Lektüre dieses Kapitels haben Sie Folgendes gelernt:

- Die Grundlagen des objektorientierten Modells
- Erzeugen und Lebensdauer von Objekten und die Kontrolle darüber
- Die drei wesentlichen Schlüsselwörter für die Zugriffsbeschränkung (`public`, `protected` und `private`)
- Die Vorteile der Klassenvererbung
- Tipps für eine erfolgreiche Ausnahmebehandlung

3.2 Objekte

Der grundlegende Unterschied zwischen objektorientierter und funktionaler Programmierung besteht darin, dass bei der OOP Daten und Quelltext in einer als *Objekt* bekannten Einheit gebündelt sind. Objektorientierte Anwendungen sind in der Regel in eine Vielzahl von Objekten aufgespaltet, die miteinander zusammenarbeiten. Normalerweise ist jedes Objekt ein Bestandteil des Problems, das sich selbst enthält und eine Anzahl von Eigenschaften und Methoden hat. Die *Eigenschaften* sind die Daten des Objekts, was hauptsächlich bedeutet, dass die Variablen zum Objekt

gehören. Die *Methoden* sind – wenn Sie von einem funktionalen Hintergrund kommen – im Wesentlichen die Funktionen, die das Objekt unterstützt. Wenn wir einen Schritt weiter gehen, versteht man unter dem *Interface* des Objekts die Funktionalität, die für den Zugriff und die Verwendung durch andere Objekte gedacht ist.

Abbildung 3.1 stellt eine Klasse dar. Eine Klasse ist eine Schablone für ein Objekt und beschreibt, welche Methoden und Eigenschaften ein Objekt dieses Typs haben wird. In diesem Beispiel stellt die Klasse eine Person dar. Sie können für jede Person der Anwendung eine eigene Instanz der Klasse erstellen, die die Informationen über diese Person darstellt. Wenn z.B. zwei Personen Joe und Judy heißen, erzeugen wir zwei getrennte Instanzen der Klasse und rufen jeweils die Methode `setName()` mit dem entsprechenden Namen auf, um die Variable `$name` zu initialisieren, die den Namen der Person enthält. Die Methoden und Mitglieder, die andere wechselwirkende Objekte nutzen können, sind die Verträge der Klasse. In diesem Beispiel bestehen die Verträge der Person mit der äußeren Welt aus den beiden `get`- und `set`-Methoden `setName()` und `getName()`.

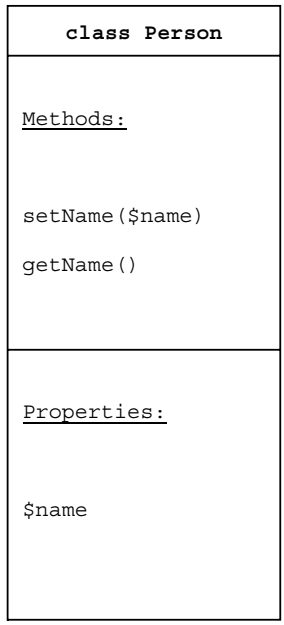

Abbildung 3.1: Diagramm der Klasse Person

Der folgende PHP-Code definiert die Klasse, erzeugt zwei Instanzen, setzt für jede Instanz einen geeigneten Namen und gibt ihn aus:

```
class Person {
    private $name;

    function setName($name)
```

```
    {
        $this->name = $name;
    }

    function getName()
    {
        return $this->name;
    }
};

$judy = new Person();
$judy->setName("Judy");

$joe = new Person();
$joe->setName("Joe");

print $judy->getName() . "\n";
print $joe->getName(). "\n";
```

3.3 Eine Klasse deklarieren

Sie haben im vorangegangenen Beispiel vermutlich gesehen, dass sich eine Klasse (eine Objekt-Schablone) relativ leicht deklarieren lässt. Sie verwenden dazu das Schlüsselwort `class`, geben der Klasse einen Namen und führen alle Methoden und Eigenschaften auf, die eine Instanz dieser Klasse haben sollte:

```
class MyClass {
    … // Liste von Methoden
    …
    … // Liste von Eigenschaften
    …
}
```

Sie haben sicherlich bemerkt, dass wir vor dem Deklarieren der Eigenschaft `$name` das Schlüsselwort `private` verwendet haben. Wir werden dieses Schlüsselwort später genauer erläutern, aber im Wesentlichen bedeutet es, dass nur Methoden in dieser Klasse einen Zugriff auf `$name` haben. Es zwingt jeden, der diese Eigenschaft setzen oder auslesen möchte, die Methoden `getName()` und `setName()` zu verwenden, die das Interface der Klasse darstellen, die von anderen Objekten oder Quelltext zu nutzen ist.

3.4 Das Schlüsselwort new und Konstruktoren

Instanzen von Klassen werden mit Hilfe des Schlüsselworts `new` erzeugt. Im vorangegangenen Beispiel erzeugten wir über `$judy = new Person();` eine neue Instanz der Klasse `Person`. Während des Aufrufs von `new` wird ein neues Objekt mit eigenen Kopien der in der angeforderten Klasse definierten Eigenschaften zugeteilt und dann

der Konstruktor des Objekts aufgerufen, sofern einer definiert wurde. Der Konstruktor ist eine Methode namens `__construct()`, die nach der Objekterstellung automatisch vom Schlüsselwort `new` aufgerufen wird. Es wird normalerweise dazu verwendet, verschiedene Initialisierungen vorzunehmen, z.B. für die Eigenschaften. Es können auch Argumente an Konstruktoren übergeben werden. In diesem Fall müssen Sie dem Konstruktor beim Schreiben der `new`-Anweisung auch die Funktionsparameter in Klammern übergeben.

In PHP 4 mussten Sie an Stelle von `__construct()`wie in C++ als Konstruktorname eine Methode mit dem Namen der Klasse definieren. Das funktioniert auch noch in PHP 5, jedoch sollten Sie für neue Anwendungen die neue vereinheitlichte Namensgebung verwenden.

Wir können das letzte Beispiel neu schreiben, um die Namen der Personen in der `new`-Zeile zu übergeben:

```
class Person {
    function __construct($name) {
        $this->name = $name;
    }

    function getName(){
        return $this->name;
    }

    private $name;
};

$judy = new Person("Judy") . "\n";
$joe = new Person("Joe") . "\n";

print $judy->getName();
print $joe->getName();
```

Dieser Code ergibt dieselbe Ausgabe wie das vorherige Beispiel.

Hinweis

Da ein Konstruktor keinen Wert zurückgeben kann, wird ein Fehler aus einem Konstruktor üblicherweise über eine Ausnahme gesendet.

3.5 Destruktoren

Destruktoren sind das Gegenteil von Konstruktoren. Sie werden bei der Zerstörung des Objekts aufgerufen (wenn es z.B. keine Referenzen mehr auf das Objekt gibt). Da PHP sicherstellt, dass am Ende jeder Anfrage alle Ressourcen freigegeben werden, ist die Bedeutung von Destruktoren beschränkt. Sie können jedoch weiterhin sinnvoll sein, um bestimmte Aktionen auszuführen, wie das Leeren einer Ressource oder das Protokollieren von Informationen über die Zerstörung eines Objekts. Es gibt zwei Situationen, in denen ein Destruktor aufgerufen werden könnte: während der Ausführung eines Scripts, wenn alle Referenzen auf ein Objekt zerstört werden, oder wenn das Ende des Scripts erreicht ist und PHP die Anfrage beendet. Die letzte Situation ist heikel, da Sie sich auf Objekte verlassen, die möglicherweise bereits ihre Destruktoren aufgerufen haben und nicht mehr erreichbar sind. Verwenden Sie sie daher mit Vorsicht und verlassen Sie sich in Ihren Destruktoren nicht auf andere Objekte.

Das Definieren eines Destruktors besteht lediglich darin, Ihrer Klasse eine Methode `__destruct()` hinzuzufügen:

```
class MyClass  {
    function __destruct()
    {
        print "An object of type MyClass is being destroyed\n";
    }
}

$obj = new MyClass();
$obj = NULL;
```

Dieses Script gibt Folgendes aus:

```
An object of type MyClass is being destroyed
```

In diesem Beispiel wird, wenn `$obj = NULL;` erreicht wird, der einzige Handle des Objekts zerstört und daher der Destruktor aufgerufen und das Objekt selbst zerstört. Der Destruktor würde sogar ohne die letzte Zeile aufgerufen werden, aber erst am Ende der Anfrage, wenn die PHP-Engine herunterfährt.

Hinweis

PHP garantiert nicht den genauen Zeitpunkt für den Aufruf des Destruktors. Er könnte einige Anweisungen später erfolgen, nachdem die letzte Referenz auf das Objekt freigegeben wurde. Achten Sie daher darauf, Ihre Anweisungen so zu schreiben, dass Sie das nicht betreffen kann.

3.6 Zugriff auf Methoden und Eigenschaften mit der Variable $this

Beim Ausführen der Methode eines Objekts wird automatisch eine spezielle Variable namens `$this` definiert, die eine Referenz auf das Objekt selbst enthält. Durch den Einsatz dieser Variablen und der Schreibweise `->` können die Methoden und Eigenschaften des Objekts weiter referenziert werden. Z.B. können Sie auf die Eigenschaft `$name` über `this->name` zugreifen. (Beachten Sie, dass vor dem Eigenschaftsnamen kein `$`-Zeichen steht.) Genauso können Sie auf eine Methode des Objekts zugreifen; z.B. können Sie aus einer der Methoden von `Person` heraus `getName()` über `this->getName` aufrufen.

3.6.1 public, protected und private für Eigenschaften

Ein Schlüsselparadigma der objektorientierten Programmierung ist die Kapselung und der Zugriffsschutz für Objekteigenschaften (Membervariablen). In den meisten objektorientierten Sprachen gibt es im Wesentlichen drei Schlüsselwörter für die Zugriffsbeschränkung: `public`, `protected` und `private`. Bei der Definition eines Klassenmembers in der Klassendefinition muss der Entwickler vor der Deklaration des Members eines dieser drei Schlüsselwörter angeben. Falls Ihnen das Objektmodell von PHP 3 oder 4 vertraut ist, werden Sie wissen, dass dort alle Klassenmember mit dem Schlüsselwort `var` definiert wurden, das `public` in PHP 5 entspricht. Aus Gründen der Abwärtskompatibilität wurde `var` beibehalten, es ist jedoch veraltet, so dass Sie Ihre Scripts auf die neuen Schlüsselwörter umstellen sollten:

```
class MyClass {
    public $publicMember = "Public member";
    protected $protectedMember = "Protected member";
    private $privateMember = "Private member";

    function myMethod(){
        // ...
    }
}

$obj = new MyClass();
```

Auf dieses Beispiel werden wir aufbauen, um die Verwendung der Zugriffsmodifizierer zu demonstrieren.

Zunächst seien die Definitionen der drei Zugriffsmodifizierer gegeben:

- `public`
  Der Zugriff auf öffentliche (public) Member kann sowohl von außerhalb eines Objekts unter Verwendung von `$obj->publicMember` als auch von innerhalb der Methode `myMethod` über die spezielle Variable `$ this` erfolgen (z.B. `$this->publicMember`). Wenn eine andere Klasse ein öffentliches Member erbt, gelten dieselben Regeln,

und es kann sowohl von außerhalb der Objekte der abgeleiteten Klasse als auch von innerhalb seiner Methoden erreicht werden.

- `protected`
  Der Zugriff auf geschützte (protected) Member kann nur aus der Methode eines Objekts heraus erfolgen – z.B. `$this->protectedMember`. Wenn eine andere Klasse ein geschütztes Member erbt, gelten dieselben Regeln, und es kann aus den Methoden des abgeleiteten Objekts heraus über die spezielle Variable `$this` erreicht werden.
- `private`
  Private (private) Member ähneln den geschützten, da der Zugriff auf sie nur aus der Methode eines Objekts heraus erfolgen kann. Sie sind jedoch nicht von den Methoden eines abgeleiteten Objekts erreichbar. Da private Eigenschaften in abgeleiteten Klassen nicht sichtbar sind, können zwei verwandte Klassen dieselben privaten Eigenschaften deklarieren. Jede Klasse sieht ihre eigene private Kopie, die mit der anderen nicht verwandt ist.

Normalerweise verwenden Sie `public` für Member, auf die Sie von außerhalb des Objekts (d.h. seine Methoden) zugreifen möchten und `private` für Member, die bezüglich der Objektlogik intern sind. Setzen Sie `protected` für Member ein, die zwar intern sind, wo es jedoch sinnvoll sein könnte, dass abgeleitete Klassen sie überschreiben:

```
class MyDbConnectionClass {
    public $queryResult;
    protected $dbHostname = "localhost";
    private $connectionHandle;

    // ...
}

class MyFooDotComDbConnectionClass extends MyDbConnectionClass {
    protected $dbHostname = "foo.com";
}
```

Dieses unvollständige Beispiel zeigt typische Verwendungen dieser drei Zugriffsmodifizierer. Diese Klasse verwaltet eine Datenbankverbindung einschließlich der Abfragen:

- Das Verbindungshandle zur Datenbank ist ein privates Member (`private`), da es nur von der internen Klassenlogik genutzt wird und Benutzer dieser Klasse nicht darauf zugreifen sollten.
- Im diesem Beispiel wird dem Benutzer der Klasse `MyDbConnectionClass` der Hostname des Datenbankrechners nicht bekannt gegeben. Um ihn zu überschreiben, kann der Entwickler von der ursprünglichen Klasse erben und ihn verändern.
- Der Entwickler sollte auf das Ergebnis der Abfrage selbst zugreifen können. Es wurde daher als öffentlich (`public`) deklariert.

Beachten Sie, dass die Zugriffsmodifizierer so entworfen wurden, dass Klassen (oder genauer ihre Interfaces zur äußeren Welt) während der Vererbung immer eine IS-A-

Beziehung aufrechterhalten. Wenn daher eine Elternklasse ein Member als öffentlich deklariert, muss das Kind es genauso tun, da es andernfalls keine IS-A-Beziehung mit dem Elternteil hätte. Das bedeutet, dass alles, was mit der Elternklasse ausführbar ist, auch für das Kind möglich ist.

3.6.2 Methoden als public, protected und private deklarieren

Zugriffsmodifizierer können auch in Verbindung mit Objektmethoden verwendet werden, wobei dieselben Regeln gelten:

- Öffentliche Methoden (`public`) können aus allen Bereichen heraus aufgerufen werden.
- Geschützte Methoden (`protected`) können nur aus einer Methode der Klasse oder einer erbenden Klasse heraus aufgerufen werden.
- Private Methoden (`private`) können nur aus einer Methode der Klasse, aber nicht aus einer erbenden Klasse heraus aufgerufen werden. Wie bei den Eigenschaften können private Methoden von der abgeleiteten Klasse neu definiert werden. Jede Klasse hat ihre eigene Version der Methode:

```
class MyDbConnectionClass {
    public function connect()
    {
        $conn = $this->createDbConnection();
        $this->setDbConnection($conn);
        return $conn;
    }

    protected function createDbConnection()
    {
        return mysql_connect("localhost");
    }

    private function setDbConnection($conn)
    {
        $this->dbConnection = $conn;
    }

    private $dbConnection;
}

class MyFooDotComDbConnectionClass extends MyDbConnectionClass {
    protected function createDbConnection()
    {
        return mysql_connect("foo.com");
    }
}
```

Dieses Grundgerüst für ein Quelltextbeispiel könnte für eine Datenbankverbindungsklasse verwendet werden. Die Methode `connect()` ist für einen Aufruf von außen gedacht. `createDbConnection()` ist eine interne Methode, ermöglicht aber eine Vererbung und Veränderung und ist daher als `protected` gekennzeichnet. Die Methode `setDbConnection()` ist vollständig intern und daher als `private` gekennzeichnet.

Hinweis

Wenn für eine Methode kein Zugriffsmodifizierer angegeben ist, wird als Standard `public` verwendet. Aus diesem Grund wird `public` in den späteren Kapiteln oftmals nicht angegeben.

3.6.3 Statische Eigenschaften

Wie Sie jetzt wissen, können Klassen Eigenschaften deklarieren. Jede Instanz einer Klasse (d.h. jedes Objekt) hat seine eigene Kopie dieser Eigenschaften. Eine Klasse kann jedoch auch *statische Eigenschaften* enthalten. Im Gegensatz zu den regulären Eigenschaften gehören sie zur Klasse selbst, aber nicht zu einer ihrer Instanzen. Im Unterschied zu den Objekt- oder Instanzeigenschaften werden sie daher oftmals *Klasseneigenschaften* genannt. Sie können sich statische Eigenschaften auch als globale Variablen vorstellen, die sich innerhalb der Klasse befinden, aber über die Klasse überall erreichbar sind.

Statische Eigenschaften werden mit Hilfe des Schlüsselworts `static` definiert:

```
class MyClass {
    static $myStaticVariable;
    static $myInitializedStaticVariable = 0;
}
```

Um auf statische Eigenschaften zuzugreifen, müssen Sie den Eigenschaftsnamen zusammen mit seiner Klasse angeben:

```
MyClass::$myInitializedStaticVariable++;
print MyClass::$myInitializedStaticVariable;
```

Dieses Beispiel gibt die Zahl `1` aus.

Wenn Sie aus einer der Methode einer Klasse auf das Member zugreifen, können Sie die Eigenschaft auch durch Voranstellen des speziellen Klassennamens `self` erreichen, der eine Kurzbezeichnung für die Klasse ist, zu der die Methode gehört:

```
class MyClass {
    static $myInitializedStaticVariable = 0;
```

```
    function myMethod()
    {
        print self::$myInitializedStaticVariable;
    }
}

$obj = new MyClass();
$obj->myMethod();
```

Dieses Beispiel gibt die Zahl `0` aus.

Sie fragen sich vermutlich, ob diese statischen Eigenschaften überhaupt sinnvoll sind.

Eine mögliche Verwendung besteht darin, allen Instanzen einer Klasse eine eindeutige Identifikationsnummer zu geben:

```
class MyUniqueIdClass {
    static $idCounter = 0;

    public $uniqueId;

    function __construct()
    {
        self::$idCounter++;
        $this->uniqueId = self::$idCounter;
    }
}

$obj1 = new MyUniqueIdClass();
print $obj1->uniqueId . "\n";
$obj2 = new MyUniqueIdClass();
print $obj2->uniqueId . "\n";
```

Dieses Beispiel gibt Folgendes aus:

```
1
2
```

Die Eigenschaftsvariable `$uniqueId` des ersten Objekts ist gleich `1` und die des zweiten gleich `2`.

Ein noch besseres Beispiel für den Einsatz statischer Eigenschaften ist das Singleton-Muster, das im nächsten Kapitel vorgestellt wird.

3.6.4 Statische Methoden

In PHP können nicht nur Eigenschaften, sondern auch Methoden als *statisch* deklariert werden. Das bedeutet, dass die statischen Methoden Bestandteile der Klasse und nicht an eine bestimmte Objektinstanz und dessen Eigenschaften gebunden sind. Daher

kann in diesen Methoden nicht auf `$this`, aber über `self` auf die Klasse selbst zugegriffen werden. Da statische Methoden nicht zu einem speziellen Objekt gehören, können Sie sie über die Syntax `class_name::method()` aufrufen, ohne eine Objektinstanz zu erzeugen. Sie können sie auch aus einer Objektinstanz heraus mit `$this->method()` aufrufen, aber `$this` ist in der aufgerufenen Methode nicht definiert. Zur Deutlichkeit sollten Sie `self::method` an Stelle von `$this->method()` verwenden.

Hier ist ein Beispiel:

```
class PrettyPrinter {
    static function printHelloWorld()
    {
        print "Hello, World";
        self::printNewline();
    }

    static function printNewline()
    {
        print "\n";
    }
}

PrettyPrinter::printHelloWorld();
```

Dieses Beispiel gibt den String `"Hello, World"`, gefolgt von einem Zeilenumbruch, aus. Obwohl das Beispiel nutzlos ist, können Sie erkennen, dass `printHelloWorld()` auf der Klasse aufgerufen werden kann, ohne eine Objektinstanz mit dem Klassennamen zu erzeugen, und die statische Methode selbst kann mit Hilfe der Schreibweise `self::` eine andere statische Methode der Klasse, `printNewline()`, aufrufen. Sie können die statische Methode einer Elternklasse über die Schreibweise `parent::` aufrufen, die später in diesem Kapitel behandelt wird.

3.7 Klassen-Konstanten

Globale Konstanten gibt es in PHP bereits seit langem. Sie konnten über die Funktion `define()` definiert werden, die in Kapitel 2, »Grundlegendes zu PHP 5« beschrieben wurde. Mit der in PHP 5 verbesserten Unterstützung für Kapselung können Sie jetzt Konstanten innerhalb von Klassen definieren. Sie gehören ähnlich wie statische Member zur Klasse und nicht zu deren Instanzen. Für Klassenkonstanten ist die Groß- und Kleinschreibung stets wichtig. Die Syntax zum Deklarieren ist einfach, und der Zugriff auf Konstanten ähnelt dem auf statische Member:

```
class MyColorEnumClass {
    const RED = "Red";
    const GREEN = "Green";
    const BLUE = "Blue";
```

```
    function printBlue()
    {
        print self::BLUE;
    }
}

print MyColorEnumClass::RED;
$obj = new MyColorEnumClass();
$obj->printBlue();
```

Dieser Code gibt `"Red"` gefolgt von `"Blue"` aus. Er zeigt die Möglichkeit auf, die Konstante sowohl aus einer Methode einer Klasse heraus über das Schlüsselwort `self`, als auch über den Klassennamen `MyColorEnumClass` aufzurufen.

Wie der Name nahe legt, sind Konstanten konstant und können nach ihrer Definition weder verändert noch entfernt werden. Übliche Verwendungszwecke für Konstanten sind die Definition von Aufzählungen wie im vorangegangenen Beispiel oder Konfigurationswerte wie der Benutzername einer Datenbank, den die Anwendung nicht ändern sollte.

Tipp

Wie globale Konstanten sollten Sie Konstanten in Großbuchstaben schreiben, da das allgemein üblich ist.

3.8 Objekte klonen

Wenn Sie ein Objekt erstellen (über das Schlüsselwort `new`), ist der Rückgabewert ein Handle für ein Objekt oder mit anderen Worten die *ID-Nummer* des Objekts. Das ist anders als in PHP 4, wo der Wert das Objekt selbst war. Das bedeutet, dass sich die Syntax für den Methodenaufruf oder den Zugriff auf Eigenschaften nicht geändert hat, wohl aber die Semantik des Kopierens von Objekten.

Betrachten Sie den folgenden Quelltext:

```
class MyClass {
    public $var = 1;
}

$obj1 = new MyClass();
$obj2 = $obj1;
$obj2->var = 2;
print $obj1->var;
```

In PHP 4 würde dieser Code `1` ausgeben, da `$obj2` der Objektwert von `$obj1` zugewiesen wird. Daher wird eine Kopie angelegt, die `$obj1` unverändert lässt. In PHP 5 jedoch wird, da `$obj1` ein Objekthandle ist (seine ID-Nummer), das Handle nach `$obj2` kopiert. Wenn also `$obj2` verändert wird, ändern Sie in der Tat dasselbe Objekt, das `$obj1` referenziert. Das Ausführen dieses Codefragments führt also zu der Ausgabe `2`.

Manchmal wollen Sie jedoch tatsächlich eine Kopie des Objekts erstellen. Wie können Sie das erreichen? Die Lösung besteht in dem Sprachkonstrukt `clone`. Dieser eingebaute Operator erzeugt automatisch eine neue Instanz des Objekts mit einer eigenen Kopie der Eigenschaften. Die Eigenschaftswerte werden so kopiert, wie sie sind. Zusätzlich können Sie eine Methode `__clone()`definieren, die auf das neu erstellte Objekt angewendet wird, um abschließende Änderungen durchzuführen.

Hinweis

Referenzen werden als Referenzen kopiert und führen keine tief gehende Kopie aus. Wenn also eine Ihrer Eigenschaften als Referenz auf eine andere Variable zeigt (nachdem sie als Referenz zugewiesen wurde), zeigt das geklonte Objekt nach dem automatischen Klonen auf dieselbe Variable.

Eine Änderung der Zeile `$obj2 = $obj1;` im vorherigen Beispiel in `$obj2 = clone $obj1;` weist `$obj2` ein Handle auf eine neue Kopie von `$obj1` zu, was die Ausgabe `1` ergibt.

Wie zuvor erwähnt, können Sie für jede Ihrer Klassen eine Methode `__clone()` erstellen. Betrachten Sie ein Objekt, das eine Ressource wie z.B. ein Dateihandle enthält. Sie möchten nicht, dass das neue Objekt auf dasselbe Dateihandle zeigt, sondern selbst eine neue Datei öffnet, so dass es seine eigene private Kopie erhält:

```
class MyFile {
    function setFileName($file_name)
    {
        $this->file_name = $file_name;
    }

    function openFileForReading()
    {
      $this->file_handle = fopen($this->file_name, "r");
    }

    function __clone()
    {
        if ($this->file_handle) {
            $this->file_handle = fopen($this->file_name, "r");
        }
    }
```

```
    private $file_name;
    private $file_handle = NULL;
}
```

Obwohl dieser Quelltext nicht komplett ist, können Sie erkennen, wie Sie den Prozess des Klonens steuern können. `$file_name` wird so, wie es ist, aus dem Originalobjekt kopiert, aber wenn das Originalobjekt ein offenes Dateihandle hat (was auf das geklonte Objekt kopiert wurde), erzeugt eine neue Kopie des Objekts seine eigene Kopie des Dateihandles, indem es die Datei selber öffnet.

3.9 Polymorphismus

Polymorphismus ist vermutlich das wichtigste Thema in der objektorientierten Programmierung. Der Einsatz von Klassen und Vererbung im Gegensatz zu einer bloßen Ansammlung von Funktionen und Daten macht es einfach, eine realitätsnahe Situation zu beschreiben. Um robusten und erweiterbaren Code zu erhalten, möchten Sie normalerweise so wenige Anweisungen zur Ablaufkontrolle (wie z.B. If-Anweisungen) wie möglich verwenden. Polymorphismus erfüllt alle diese Anliegen und noch mehr.

Betrachten Sie den folgenden Quelltext:

```
class Cat {
    function miau()
    {
        print "miau";
    }
}

class Dog {
    function wuff()
    {
        print "wuff";
    }
}

function printTheRightSound($obj)
{
    if ($obj instanceof Cat) {
        $obj->miau();
    } else if ($obj instanceof Dog) {
        $obj->wuff();
    } else {
        print "Error: Passed wrong kind of object";
    }
    print "\n";
}
```

```
printTheRightSound(new Cat());
printTheRightSound(new Dog());
```

Die Ausgabe lautet

```
miau
wuff
```

Sie können leicht sehen, dass dieses Beispiel nicht erweiterbar ist. Nehmen wir an, Sie möchten es um die Geräusche von drei anderen Tieren erweitern. Sie müssten der Funktion `printTheRightSound()` drei zusätzliche `else if`-Blöcke hinzufügen, in denen Sie überprüfen, ob das aktuelle Objekt eine Instanz dieser drei neuen Tiere ist, und dann müssten Sie den Code zum Aufruf der Methoden für die Tiergeräusche hinzufügen.

Polymorphismus mit Vererbung löst dieses Problem. Es bietet die Möglichkeit, alle Methoden und Eigenschaften von der Elternklasse zu erben, und erzeugt damit eine IS-A-Beziehung.

In dem folgenden Beispiel erstellen wir eine Klasse namens `Animal`, von der alle anderen Tierklassen erben, und somit IS-A-Beziehungen der gewünschten Art, wie z. B. `Dog` mit der Elternklasse (oder dem Vorfahren) `Animal`.

Eine Vererbung erfolgt mit dem Schlüsselwort `extends`:

```
class Child extends Parent {
    ...
}
```

Und so würden Sie das vorangegangene Beispiel mit Vererbung schreiben:

```
class Animal {
    function makeSound()
    {
        print "Error: This method should be re-implemented in the children";
    }
}

class Cat extends Animal {
    function makeSound()
    {
        print "miau";
    }
}

class Dog extends Animal {
    function makeSound()
    {
        print "wuff";
```

```
    }
}

function printTheRightSound($obj)
{
    if ($obj instanceof Animal) {
        $obj->makeSound();
    } else {
        print "Error: Passed wrong kind of object";
    }
    print "\n";
}

printTheRightSound(new Cat());
printTheRightSound(new Dog());
```

Die Ausgabe lautet

```
miau
wuff
```

Sie können sehen, dass, egal wie viele Tierarten Sie diesem Beispiel hinzufügen, Sie keine Änderungen an `printTheRightSound()` vornehmen müssen, da die Überprüfung `instanceof Animal` alle abdeckt, ebenso wie der Aufruf von `$obj->makeSound()`.

Dieses Beispiel kann noch weiter verbessert werden. Gewisse in PHP verfügbare Modifizierfaktoren bieten eine stärkere Kontrolle über den Vererbungsprozess. Sie werden später in diesem Kapitel ausführlich behandelt. Z.B. können die Klasse `Animal` und ihre Methode `makeSound()` als `abstract` gekennzeichnet werden, was nicht nur bedeutet, dass Sie keine bedeutungslose Implementierung der Definition von `makeSound()` in der Klasse `Animal` vornehmen müssen, sondern auch jede erbende Klasse dazu zwingt, sie zu implementieren. Darüber hinaus können wir Zugriffsmodifizierer wie `public` für die Methode `makeSound()` angeben, was bedeutet, dass sie von überall aufgerufen werden kann.

Hinweis

PHP unterstützt keine mehrfache Vererbung wie C++. Es enthält mit Java-ähnlichen Interfaces eine andere Lösung, um mehr als eine IS-A-Beziehung für eine gegebene Klasse zu erstellen, die später in diesem Kapitel behandelt wird.

3.10 parent:: und self::

PHP unterstützt zwei reservierte Klassennamen, die das Schreiben von objektorientierten Anwendungen erleichtern. `self::` bezieht sich auf die aktuelle Klasse und wird normalerweise für den Zugriff auf statische Member, Methoden und Konstanten verwendet. `parent::` bezieht sich auf die Elternklasse und wird meistens dann genutzt, wenn der Konstruktor oder Methoden der Elternklasse aufgerufen werden sollen. Es kann auch für den Zugriff auf Member und Konstanten verwendet werden. Sie sollten `parent::` an Stelle des Klassennamens der Elternklasse verwenden, da Änderungen in der Klassenhierarchie erleichtert werden, wenn die Namen nicht hart codiert sind.

Das folgende Beispiel nutzt sowohl `parent::` als auch `self::` zum Zugriff auf die Klassen `Child` und `Ancestor`:

```
class Ancestor {
    const NAME = "Ancestor";
    function __construct()
    {
        print "In " . self::NAME . " constructor\n";
    }
}

class Child extends Ancestor {
    const NAME = "Child";
    function __construct()
    {
        parent::__construct();
        print "In " . self::NAME . " constructor\n";
    }
}

$obj = new Child();
```

Es erzeugt die folgende Ausgabe:

```
In Ancestor constructor
In Child constructor
```

Stellen Sie sicher, dass Sie wo immer möglich diese beiden Klassennamen verwenden.

3.11 Der Operator instanceof

Der Operator `instanceof` wurde als einfachere Alternative an Stelle der bereits vorhandenen, eingebauten Funktion `is_a()` geschaffen (die jetzt veraltet ist). Im Gegensatz zu letzterer wird `instanceof` als logischer Binäroperator verwendet:

```
class Rectangle {
    public $name = __CLASS__;
}

class Square extends Rectangle {
    public $name = __CLASS__;
}

class Circle {
    public $name = __CLASS__;
}

function checkIfRectangle($shape)
{
    if ($shape instanceof Rectangle) {
        print $shape->name;
        print " is a rectangle\n";
    }
}

checkIfRectangle(new Square());
checkIfRectangle(new Circle());
```

Dieses kleine Programm gibt `"Square is a rectangle\n"` aus. Beachten Sie die Verwendung von `__CLASS__`. Es ist eine spezielle Konstante, die den Namen der aktuellen Klasse auflöst. Wie zuvor erwähnt, ist `instanceof` ein Operator und kann daher in Ausdrücken zusammen mit anderen Operatoren (z.B. dem Negationsoperator `!`) eingesetzt werden. Damit können Sie auf einfache Weise eine Funktion `checkIfNotRectangle` schreiben:

```
function checkIfNotRectangle($shape)
{
    if (!($shape instanceof Rectangle)) {
        print $shape->name;
        print " is not a rectangle\n";
    }
}
```

Hinweis

`instanceof` prüft auch, ob ein Objekt ein Interface implementiert (was auch eine klassische IS-A-Beziehung ist). Interfaces werden später in diesem Kapitel behandelt.

3.12 Abstrakte Methoden und Klassen

Beim Entwerfen von Klassenhierarchien kann es wünschenswert sein, gewisse Methoden anzugeben, die erst die abgeleitete Klasse implementieren soll. Nehmen wir an, Sie haben eine Klassenhierarchie wie in Abbildung 3.2:

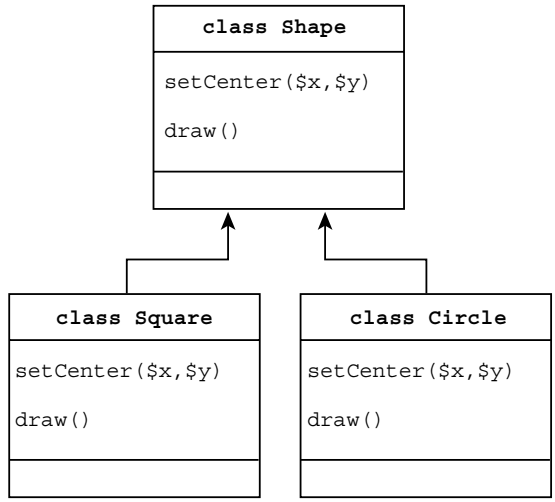

Abbildung 3.2: Klassenhierarchie

Es kann sinnvoll sein, `setCenter($x, $y)` in der Klasse `Shape` zu implementieren und die Implementierung der Methoden `draw()` den konkreten Klassen `Square` und `Circle` zu überlassen. In diesem Fall müssen Sie die Methode `draw()` als abstrakte (`abstract`) Methode deklarieren, so dass PHP weiß, dass Sie nicht beabsichtigen, sie in der Klasse `Shape` zu implementieren. Die Klasse `Shape` wird dann eine abstrakte (`abstract`) Klasse genannt, was bedeutet, dass sie keine vollständige Funktionalität hat, sondern nur zum Erben dient. Sie können keine Instanzen abstrakter Klassen bilden. Sie können beliebig viele Methoden als `abstract` definieren, doch sobald mindestens eine Methode einer Klasse `abstract` ist, muss auch die gesamte Klasse als `abstract` deklariert werden. Diese doppelte Definition bietet Ihnen die Option, eine Klasse auch dann als `abstract` zu definieren, wenn sie keine abstrakten Methoden hat, und zwingt Sie dazu, eine Klasse mit abstrakten Methoden als `abstract` zu definieren, so dass anderen deutlich wird, was Sie gemeint haben.

Das vorherige Klassendiagramm kann in den folgenden PHP-Quelltext übersetzt werden:

```
abstract class Shape {
    function setCenter($x, $y) {
        $this->x = $x;
        $this->y = $y;
```

```
    }

    abstract function draw();

    protected $x, $y;
}

class Square extends Shape {
    function draw()
    {
        // Hier steht der Code, der das Quadrat zeichnet.
        ...
    }
}

class Circle extends Shape {
    function draw()
    {
        // Hier steht der Code, der den Kreis zeichnet.
        ...
    }
}
```

Wie Sie sehen, enthält die abstrakte Methode `draw()` keinerlei Code.

Hinweis

Im Gegensatz zu einigen anderen Sprachen können Sie keine abstrakte Methode mit einer Standardimplementierung installieren. In PHP ist eine Methode entweder `abstract` (ohne Code) oder vollständig definiert.

3.13 Interfaces

Die **Klassenvererbung** bietet die Möglichkeit, eine Eltern-Kind-Beziehung zwischen Klassen zu beschreiben. Z.B. können Sie eine Basisklasse `Form` haben, von der Sie sowohl `Square` als auch `Circle` ableiten. Oft besteht jedoch der Bedarf, zusätzliche »Interfaces« zu Klassen hinzuzufügen, womit weitere Verträge gemeint sind, die die Klasse erfüllen muss. In C++ erreicht man das über mehrfache Vererbung und Ableitung von zwei Klassen. PHP wählt Interfaces als Alternative zur mehrfachen Vererbung, was die Möglichkeit bietet, zusätzliche Verträge anzugeben, denen die Klasse folgen muss. Ein Interface wird ähnlich wie eine Klasse deklariert, enthält jedoch nur Prototypen von Funktionen (ohne Implementierungen) und Konstanten. Jede Klasse, die dieses Interface »implementiert«, enthält automatisch die dort definierten Kons-

tanten und muss als die implementierende Klasse die Funktionsdefinitionen für die Prototypen des Interfaces liefern. Sie alle sind abstrakte Methoden (sofern Sie die implementierende Klasse nicht als `abstract` definieren).

Ein Interface wird mit der folgenden Syntax implementiert:

```
class A implements B, C, ... {
    ...
}
```

Klassen, die ein Interface implementieren, haben zu ihm eine `instanceof`-Beziehung (IS-A-Beziehung). Wenn z. B. die Klasse `A` das Interface `myInterface` implementiert, gibt das folgende Codefragment `'$obj is-A myInterface'` aus:

```
$obj = new A();
if ($obj instanceof myInterface) {
    print '$obj is-A myInterface';
}
```

Das folgende Beispiel definiert ein Interface namens `Loggable`, das Klassen implementieren können, um anzugeben, was die Funktion `MyLog()` protokolliert. Objekte von Klassen, die dieses Interface nicht implementieren und an die Funktion `MyLog()`übergeben werden, erzeugen eine Fehlermeldung:

```
interface Loggable {
    function logString();
}

class Person implements Loggable {
    private $name, $address, $idNumber, $age;
    function logString() {
        return "class Person: name = $this->name, ID = $this->idNumber\n";
    }
}

class Product implements Loggable {
    private $name, $price, $expiryDate;
    function logString() {
        return "class Product: name = $this->name, price = $this->price\n";
    }
}

function MyLog($obj) {
    if ($obj instanceof Loggable) {
        print $obj->logString();
    } else {
        print "Error: Object doesn't support Loggable interface\n";
    }
}
```

```
$person = new Person();
// ...
$product = new Product();

MyLog($person);
MyLog($product);
```

Hinweis

Interfaces werden stets als `public` betrachtet; daher können Sie beim Deklarieren keine Zugriffsmodifizierer für die Prototypen der Methoden angeben.

Achtung

Sie dürfen keine mehrfachen Interfaces erstellen, die miteinander kollidieren (z.B. Interfaces, die dieselben Konstanten oder Methoden definieren).

3.14 Interfaces vererben

Interfaces können von anderen Interfaces erben. Die Syntax ist ähnlich der von Klassen, ermöglicht jedoch mehrfache Vererbung:

```
interface I1 extends I2, I3, ... {
    ...
}
```

Ähnlich dem Fall, wenn Klassen Interfaces implementieren, kann ein Interface nur dann ein anderes erweitern, wenn beide nicht miteinander kollidieren. (Das bedeutet, Sie erhalten einen Fehler, wenn `I2` Methoden oder Konstanten definiert, die bereits `I1` angegeben hat.)

3.15 final-Schlüsselworte

Bisher wissen Sie, dass geerbte Methoden beim Erweitern einer Klasse (oder Erben von einer Klasse) mit einer neuen Implementierung überschrieben werden können.

Es gibt jedoch Situationen, in denen Sie sicherstellen möchten, dass eine Methode in einer abgeleiteten Klasse nicht neu implementiert werden kann. Für diesen Zweck unterstützt PHP den Java-ähnlichen Zugriffsmodifizierer `final`. Er markiert eine Methode als die endgültige Version, die nicht überschrieben werden kann.

Das folgende Beispiel ist kein gültiges PHP-Script, da es versucht, eine `final`-Methode zu überschreiben:

```
class MyBaseClass {
    final function idGenerator()
    {
        return $this->id++;
    }

    protected $id = 0;
}

class MyConcreteClass extends MyBaseClass {
    function idGenerator()
    {
        return $this->id += 2;
    }
}
```

Dieses Script funktioniert nicht, da die Definition von `idGenerator()` als `final` in der Klasse `MyBaseClass` der abgeleiteten Klasse verbietet, die Methode zu überschreiben und das Verhalten der Logik zur ID-Erstellung zu ändern.

3.16 final

Ähnlich wie Methoden können Sie auch Klassen als `final` definieren. Damit verbieten Sie das Erben von dieser Klasse. Der folgende Code funktioniert nicht:

```
final class MyBaseClass {
    ...
}

class MyConcreteClass extends MyBaseClass {
    ...
}
```

`MyBaseClass` wurde als `final` deklariert, so dass `MyConcreteClass` sie nicht erweitern darf und das Script fehlschlägt.

3.17 Die Methode __tostring()

Betrachten Sie den folgenden Code:

```
class Person {
    function __construct($name)
    {
        $this->name = $name;
    }

    private $name;
}

$obj = new Person("Andi Gutmans");

print $obj;
```

Er gibt Folgendes aus:

```
Object id #1
```

Im Gegensatz zu anderen Datentypen ist die Ausgabe der Objekt-ID in der Regel nicht interessant. Außerdem beziehen Objekte sich oft auf Daten, die eine Ausgabesemantik haben sollten – z. B. macht es für die Ausgabe eines Objekts, das eine Person darstellt, Sinn, die personenbezogenen Daten auszugeben.

Zu diesem Zweck bietet PHP die Möglichkeit, eine Funktion namens `__toString()`zu implementieren, die die String-Darstellung des Objekts zurückgibt. Wenn sie definiert ist, ruft der Befehl `print` sie auf und gibt den zurückgegebenen String aus.

Das vorige Beispiel kann mit `__toString()` in eine nützlichere Form gebracht werden:

```
class Person {
    function __construct($name)
    {
        $this->name = $name;
    }

    function __toString()
    {
        return $this->name;
    }

    private $name;

}

$obj = new Person("Andi Gutmans");

print $obj;
```

Es gibt das Folgende aus:

```
Andi Gutmans
```

Gegenwärtig rufen nur die Sprachkonstrukte `print` und `echo` die Methode `__toString()` auf. In Zukunft wird sie vermutlich von allgemeinen String-Operationen unterstützt, wie z.B. der Verkettung und der expliziten Umwandlung in einen String.

3.18 Ausnahmebehandlung

Die *Ausnahmebehandlung* ist eine der problematischeren Seiten der Softwareentwicklung. Es ist für den Entwickler nicht nur schwer zu entscheiden, was das Programm in einem Fehlerfall tun soll (wie z.B. einem Datenbank-, Netzwerk- oder Softwarefehler), sondern auch schwer, all die Stellen im Quelltext zu finden, wo Fehlerprüfungen vorzunehmen sind und die richtige Funktion zu ihrer Behandlung aufzurufen. Eine noch kompliziertere Aufgabe besteht darin, nach der Behandlung des Fehlers den Ablauf des Programms an einem bestimmten Punkt fortzusetzen.

Heute unterstützen die meisten Sprachen eine Variante des populären Paradigmas `try/catch/throw` zur Fehlerbehandlung. `try/catch` ist ein umschließendes Sprachkonstrukt, das den eingeschlossenen Quelltext schützt und der Sprache im Wesentlichen sagt »Ich kümmere mich um Ausnahmen, die in diesem Code auftreten.« Ausnahmen oder Fehler werden nach ihrer Entdeckung ausgesendet (engl. »throw«), und die Laufzeitumgebung der Sprache durchsucht den aufrufenden Verarbeitungsstapel nach einem passenden `try/catch`, das sich um die Ausnahme kümmern kann.

Dieses Verfahren hat viele Vorteile. So müssen Sie nicht überall dort, wo ein Fehler auftreten könnte, eine `if`-Anweisung anbringen; daher schreiben Sie im Endeffekt deutlich weniger Zeilen. Stattdessen können Sie den gesamten Codeabschnitt in einen `try/catch`-Block einschließen, und einen Fehler behandeln, wenn er auftritt. Nachdem Sie mit der `throw`-Anweisung einen Fehler festgestellt haben, können Sie auf einfache Weise an einen Punkt im Code zurückkehren, der für das Verarbeiten und Fortsetzen der Programmausführung zuständig ist, da `throw` den Aufrufstapel der Funktion abwickelt, bis es einen geeigneten `try/catch`-Block entdeckt.

Die Syntax von `try/catch` ist wie folgt:

```
try {
    ... // Code, der eine Ausnahme senden kann
} catch (FirstExceptionClass $exception) {
    ... // Code, der diese Ausnahme behandelt
} catch (SecondExceptionClass $exception) {
}
```

Das Konstrukt `try {}` schließt den Code ein, der eine Ausnahme senden kann, und wird von einer Reihe von `catch`-Ausdrücken gefolgt, von denen jede angibt, welche

Ausnahmeklasse sie behandelt und unter welchem Variablennamen die Ausnahme innerhalb des `catch`-Blocks erreichbar sein sollte.

Nach dem Senden einer Ausnahme wird das erste `catch()` erreicht und ein `instanceof`-Vergleich mit der deklarierten Klasse durchgeführt. Ist das Ergebnis `true`, wird der `catch`-Block betreten und die Ausnahme unter dem angegebenen Variablennamen verfügbar gemacht. Ist es `false`, wird die nächste überprüft. Sobald eine `catch`-Anweisung betreten wird, werden die weiteren ignoriert, auch wenn der `instanceof`-Vergleich `true` ergeben würde. Passt keine `catch`-Anweisung, sucht die Sprachengine weitere einschließende `try/catch`-Anweisungen in derselben Funktion. Gibt es keine, sucht sie weiter durch Abwickeln des Aufrufstapels nach den aufrufenden Funktionen.

Die `throw`-Anweisung

```
throw <object>;
```

kann nur ein Objekt aussenden, hingegen keine grundlegenden Datentypen wie Strings oder Ganzzahlen. Es gibt eine vordefinierte Ausnahmeklasse namens `Exception`, von der alle Ausnahmeklassen abgeleitet sein müssen. Der Versuch, ein Objekt auszusenden, das nicht von der Klasse `Exception` abstammt, führt zu einem Laufzeitfehler.

Das folgende Codebeispiel zeigt das Interface dieser eingebauten Ausnahmeklasse (die eckigen Klammern in der Deklaration des Konstruktors stellen optionale Parameter dar und sind keine gültige PHP-Syntax):

```
class Exception {
    function __construct([$message [,$code]]);

    final public getMessage();
    final public getCode();
    final public getFile();
    final public getLine();
    final public getTrace();
    final public getTraceAsString();

    protected $message;
    protected $code;
    protected $file;
    protected $line;
}
```

Nachfolgend ein vollständiges Beispiel zur Fehlerbehandlung:

```
class NullHandleException extends Exception {
    function __construct($message) {
        parent::__construct($message);
    }
```

```
}

function printObject($obj)
{
    if ($obj == NULL) {
         throw new NullHandleException("printObject received NULL object");
    }
    print $obj . "\n";
}

class MyName {
    function __construct($name)
    {
        $this->name = $name;
    }

    function __toString()
    {
       return $this->name;
    }

    private $name;
}

try {
    printObject(new MyName("Bill"));
    printObject(NULL);
    printObject(new MyName("Jane"));
} catch (NullHandleException $exception) {
    print $exception->getMessage();
    print " in file " . $exception->getFile();
    print " on line " . $exception->getLine() . "\n";
} catch (Exception $exception) {
    // Wird nicht erreicht
}
```

Die Ausgabe dieses Scripts ergibt:

```
Bill
printObject received NULL object in file C:\projects\php5\tests\test.php on line
 12
```

Beachten Sie, dass nicht der Name Jane, sondern nur Bill ausgegeben wird. Das liegt daran, dass die Zeile `printObject(NULL)` eine Ausnahme innerhalb der Funktion aussendet, und Jane daher ausgelassen wird. Im `catch`-Block werden geerbte Methoden wie `getFile()` verwendet, um zusätzliche Angaben darüber auszugeben, wo die Ausnahme auftrat.

Hinweis

Sie haben vermutlich bemerkt, dass der Konstruktor von `NullHandleException` seinen Elternkonstruktor aufruft. Wird der Konstruktor von `NullHandleException` weggelassen, ruft `new` standardmäßig den Elternkonstruktor auf. Es ist jedoch ein guter Stil, einen Konstruktor einzufügen und den Elternkonstruktor aufzurufen, so dass Sie es nicht vergessen, wenn Sie sich plötzlich dafür entscheiden, einen eigenen Konstruktor hinzuzufügen.

Die meisten internen Methoden senden heutzutage keine Ausnahmen aus, um die Abwärtskompatibilität zu PHP 4 zu wahren. Das beschränkt in gewisser Weise ihre Einsatzmöglichkeit, doch Sie können sie in Ihrem eigenen Code verwenden. Einige neuere Erweiterungen in PHP 5 – vor allem die objektorientierten – senden Ausnahmen. Um sicherzugehen, sollten Sie die zugehörige Dokumentation lesen.

Tipp

Folgen Sie beim Einsatz von Ausnahmen diesen grundlegenden Regeln (sowohl aus Gründen der Leistungsfähigkeit als auch zur besseren Quelltextverwaltung):

1. Denken Sie daran, dass Ausnahmen Ausnahmen sind. Sie sollten sie nur verwenden, um Probleme zu verarbeiten, was uns zur nächsten Regel bringt ...
2. Setzen Sie Ausnahmen niemals zur Ablaufsteuerung ein. Das erschwert das Lesen des Quelltexts (ähnlich wie die `goto`-Anweisungen einiger Sprachen) und ist langsam.
3. Die Ausnahme sollte nur die Angaben zum Fehler und keine Parameter (oder zusätzliche Informationen) enthalten, die die Ablaufsteuerung und die Logik innerhalb des `catch`-Blocks beeinflussen.

3.19 __autoload()

Beim Verfassen von objektorientiertem Quelltext ist es oftmals üblich, jede Klasse in eine eigene Quelldatei zu schreiben. Der Vorteil dabei ist, dass so wesentlich leichter herauszufinden ist, wo sich eine Klasse befindet. Außerdem wird der Umfang des Quelltexts minimiert, da Sie nur die erforderlichen Klassen einzubetten brauchen. Der Nachteil ist jedoch, dass Sie oftmals Unmengen von Quelldateien einbetten müssen, was ein Ärgernis und Wartungsproblem darstellen kann und häufig dazu führt, dass zu viele Dateien eingebunden werden. `__autoload()` löst dieses Problem, indem es keine Angabe der benötigten Klassen erfordert. Wenn eine Funktion `__autoload()` definiert ist (pro Anwendung kann es nur eine solche Funktion geben) und Sie auf

eine Klasse zugreifen, die noch nicht definiert wurde, wird sie mit dem Klassennamen als Parameter aufgerufen. Das gibt Ihnen die Möglichkeit, die Klasse genau rechtzeitig einzubetten. Wenn Sie sie erfolgreich eingebunden haben, fährt Ihr Code mit der Ausführung fort, als ob sie definiert wäre. Wenn Sie die Klasse nicht erfolgreich einbetten, erzeugt die Scriptengine einen schwerwiegenden Fehler (Fatal Error), dass die Klasse nicht existiert.

Hier ist ein typisches Beispiel mit `__autoload()`:

```
MyClass.php:

<?php

class MyClass {
    function printHelloWorld()
    {
        print "Hello, World\n";
    }
}

?>


general.inc:

<?php

function __autoload($class_name)
{
    require_once($_SERVER["DOCUMENT_ROOT"] . "/classes/$class_name.php");
}

?>


main.php:

<?php

require_once "general.inc";

$obj = new MyClass();
$obj->printHelloWorld();

?>
```

Hinweis

Dieses Beispiel lässt die öffnenden und schließenden PHP-Tags nicht weg (wie andere in Kapitel 2 gezeigte Beispiele), da es sich über mehrere Dateien erstreckt und daher kein Codefragment ist.

Solange sich die Klasse `MyClass.php` im Verzeichnis `classes/` im Document-Root des Webservers befindet, lautet die Ausgabe

```
Hello, World
```

Beachten Sie, dass `MyClass.php` in `main.php` nicht explizit eingebettet wurde, sondern implizit durch den Aufruf an `__autoload()`. Sie halten die Definition von `__autoload()` normalerweise in einer Datei, die von allen Ihrer Haupt-Scriptdateien eingebunden wird (ähnlich `general.inc` in diesem Beispiel). Wenn die Anzahl Ihrer Klassen wächst, werden die Einsparungen an Quelltext und Wartung enorm sein.

Hinweis

Obwohl es bei Klassen in PHP nicht auf die Groß- und Kleinschreibung ankommt, wird die Schreibweise beim Senden einer Klasse an `__autoload()` beibehalten. Wenn Sie es vorziehen, dass die Groß- und Kleinschreibung für Dateinamen Ihrer Klassen relevant ist, sollten sie sicherstellen, dass Sie im Code konsistent sind und für Ihre Klassen stets die korrekte Schreibweise verwenden. Wenn Sie das nicht wünschen, können die den Klassennamen vor dem Einbindungsversuch mit der Funktion `strtolower()` klein schreiben, und die Klassen mit kleingeschriebenen Dateinamen speichern.

3.20 Hinweise auf Klassentypen in Funktionsparametern

Obwohl PHP keine streng typisierte Sprache ist, in der Sie angeben müssen, welchen Typ die Variablen haben, können Sie, wenn Sie möchten, die Klasse angeben, die Sie in den Parametern Ihrer Funktion oder Methode erwarten.

Hier ist der Quelltext einer typischen PHP-Funktion, die einen Funktionsparameter erhält und zunächst überprüft, ob er zur erforderlichen Klasse gehört:

```
function onlyWantMyClassObjects($obj)
{
    if (!($obj instanceof MyClass)) {
        die("Only objects of type MyClass can be sent to this function");
```

```
    }
    ...
}
```

Das Verfassen von Code, der den Typ des Objekts in jeder relevanten Funktion überprüft, kann viel Arbeit sein. Um Zeit zu sparen, können Sie die Klasse des Parameters vor dem Parameter selbst angeben.

Nachfolgend dasselbe Beispiel mit einem Hinweis auf Klassentypen:

```
function onlyWantMyClassObjects(MyClass $obj)
{
    // ...
}
```

Nach dem Aufruf und vor dem Ausführen der Funktion führt PHP automatisch eine `instanceof`-Prüfung durch. Schlägt sie fehl, bricht PHP mit einem Fehler ab. Weil die Überprüfung eine `instanceof`-Prüfung ist, ist es erlaubt, jedes Objekt zu senden, das eine IS-A-Beziehung zu dem Klassentyp unterhält. Diese Eigenschaft ist vor allem während der Entwicklung sinnvoll, da sie sicherstellt, dass Sie keine Objekte an Funktionen übergeben, die nicht für den Umgang damit gedacht sind.

3.21 Zusammenfassung

Dieses Kapitel behandelte das Objektmodell von PHP 5, einschließlich der Konzepte von Klassen und Objekten, Polymorphismus und anderer wichtiger objektorientierter Konzepte und Semantik. Wenn PHP für Sie neu ist, Sie aber bereits Programme in objektorientierten Programmiersprachen erstellt haben, werden Sie vermutlich nicht verstehen, wie Leute es bisher geschafft haben, objektorientierten Code zu schreiben. Wenn Sie bereits objektorientierten Code in PHP 4 erstellt haben, konnten Sie diese neuen Funktionen vermutlich gar nicht erwarten.