Der Typ object

`object` (`System.Object`) ist die Ausgangsbasisklasse für alle Typen. Jeder Typ kann per Upcast in ein `object` umgewandelt werden.

Um zu zeigen, wie das nützlich sein kann, stellen Sie sich einen allgemeinen *Stack* vor. Ein Stack ist eine Datenstruktur, die auf dem Prinzip »Last in, first out« (kurz LIFO) basiert. Ein Stack hat zwei Operationen: per *Push* ein Objekt auf den Stack schieben und per *Pop* eines wieder herunterholen. So sieht eine einfache Implementierung aus, die bis zu zehn Objekte verwalten kann:

```
public class Stack
{
  int position;
  object[] data = new object[10];
  public void Push (object o) { data[position++] = o; }
  public object Pop() { return data[--position]; }
}
```

Da `Stack` mit dem Typ `object` arbeitet, können wir Instanzen von *jedem beliebigen Typ* per `Push` und `Pop` auf den Stack bringen und wieder von ihm herunterbringen:

```
Stack stack = new Stack();
stack.Push ("Würstchen");
string s = (string) stack.Pop();   // Cast nötig
Console.WriteLine (s);             // Würstchen
```

`object` ist ein Referenztyp, da es eine Klasse ist. Trotzdem können Werttypen wie `int` auf `object` gecastet werden. Das ermöglicht die CLR mit einigen speziellen Operationen, die die grundlegenden Unterschiede zwischen Wert- und Referenztypen überbrücken. Man bezeichnet diese Schritte als *Boxing* und *Unboxing*.

Im Abschnitt »Generics« auf Seite 101 werden wir beschreiben, wie wir unsere `Stack`-Klasse verbessern können, damit sie besser mit Stacks von Elementen desselben Typs umgeht.

Boxing und Unboxing

Boxing ist das Casten einer Instanz eines Werttyps auf eine Instanz eines Referenztyps. Der Referenztyp kann entweder von der Klasse `object` oder ein Interface sein (siehe »Interfaces« auf Seite 94). In diesem Beispiel *boxen* wir ein `int` in ein `object`:

```
int x = 9;
object obj = x;           // int verpacken
```

Beim Unboxing wird die Operation umgekehrt, indem das Objekt zurück in den ursprünglichen Werttyp gewandelt wird:

```
int y = (int)obj;         // int auspacken
```

Unboxing erfordert einen expliziten Cast. Die Laufzeitumgebung prüft, ob der angegebene Werttyp dem tatsächlichen Objekttyp entspricht, und wirft bei Nichtübereinstimmung eine `InvalidCast-Exception`. So wird zum Beispiel hier eine Exception geworfen, weil `long` nicht genau zu `int` passt:

```
object obj = 9;       // 9 wird als int erkannt
long x = (long) obj;  // InvalidCastException
```

Der folgende Code wird aber erfolgreich sein:

```
object obj = 9;
long x = (int) obj;
```

Und dieser auch:

```
object obj = 3.5;      // Erschlossener Typ von 3.5 ist double
int x = (int) (double) obj;    // x is now 3
```

Im letzten Beispiel führt `(double)` ein *Unboxing* durch und `(int)` eine *numerische Umwandlung*.

Beim Boxing wird die Werttypinstanz in das neue Objekt *kopiert*, und beim Unboxing wird der Inhalt des Objekts wieder in eine Werttypinstanz *kopiert*:

```
int i = 3;
object boxed = i;
i = 5;
Console.WriteLine (boxed);    // 3
```

Statische und Laufzeit-Typprüfung

C# prüft Typen sowohl statisch (beim Kompilieren) als auch zur Laufzeit.

Die statische Typprüfung ermöglicht dem Compiler, die Korrektheit Ihres Programms zu prüfen, ohne es auszuführen. Der folgende Code wird fehlschlagen, da der Compiler die statische Typprüfung durchführt:

```
int x = "5";
```

Die Laufzeit-Typprüfung wird von der CLR durchgeführt, wenn Sie über eine Referenzumwandlung oder Unboxing einen Downcast vornehmen:

```
object y = "5";
int z = (int) y;       // Laufzeitfehler, Downcast
                       // fehlgeschlagen
```

Die Laufzeit-Typprüfung ist möglich, weil jedes Objekt auf dem Heap intern ein kleines Typ-Token mit abspeichert. Dieses Token kann durch den Aufruf der Methode `GetType` von `object` genutzt werden.

Die GetType-Methode und der typeof-Operator

Alle Typen in C# werden zur Laufzeit durch eine Instanz von `System.Type` repräsentiert. Ein Objekt vom Typ `System.Type` kann man auf zweierlei Weise erhalten: durch einen Aufruf von `GetType` für die Instanz oder durch den Einsatz des Operators `typeof` für einen Typnamen. `GetType` wird zur Laufzeit ausgewertet, während `typeof` statisch beim Kompilieren ermittelt wird.

`System.Type` hat Eigenschaften für den Namen des Typs, die Assembly, den Basistyp und so weiter:

```
int x = 3;

Console.Write (x.GetType().Name);               // Int32
Console.Write (typeof(int).Name);               // Int32
Console.Write (x.GetType().FullName);    // System.Int32
Console.Write (x.GetType() == typeof(int));     // True
```

System.Type stellt auch Methoden zur Verfügung, die als Brücke zum Reflection-Modell der Laufzeitumgebung dienen.

Die Member von object

Das hier sind alle Member von object:

```
public extern Type GetType();
public virtual bool Equals (object obj);
public static bool Equals (object objA, object objB);
public static bool ReferenceEquals (object objA,
                                    object objB);
public virtual int GetHashCode();
public virtual string ToString();
protected override void Finalize();
protected extern object MemberwiseClone();
```

Equals, ReferenceEquals und GetHashCode

Die Methode Equals entspricht dem Operator ==, nur dass Equals virtuell, == aber statisch ist. Das folgende Beispiel zeigt den Unterschied:

```
object x = 3;
object y = 3;
Console.WriteLine (x == y);        // False
Console.WriteLine (x.Equals (y));  // True
```

Da x und y auf den Typ object gecastet wurden, bindet der Compiler statisch gegen den Operator == von object, der die *Referenztyp*-Semantik nutzt, um zwei Instanzen zu vergleichen. (Da x und y geboxt sind, handelt es sich um zwei verschiedene Speicherplätze, die damit nicht gleich sind.) Die virtuelle Methode Equals greift auf Equals des Typs Int32 zurück, das eine *Werttyp*-Semantik beim Vergleich zweier Werte nutzt.

Die statische Methode object.Equals ruft einfach die virtuelle Methode Equals auf – nachdem sie sichergestellt hat, dass die Argumente nicht null sind.

```
object x = null, y = 3;
bool error = x.Equals (y);        // Laufzeitfehler!
bool ok = object.Equals (x, y);   // OK (false)
```

`ReferenceEquals` erzwingt einen Test auf Referenztypgleichheit (das ist gelegentlich bei Referenztypen nützlich, wenn der Operator `==` überladen wurde, um etwas anderes zu bewirken).

`GetHashCode` gibt einen Hashcode aus, der für den Einsatz in Hashtable-basierten Dictionaries geeignet ist, d. h. `System.Collections.Generic.Dictionary` und `System.Collections.Hashtable`.

Um die Gleichheitssemantik eines Typs anzupassen, müssen Sie mindestens `Equals` und `GetHashCode` überschreiben. Sie werden meist aber auch die Operatoren `==` und `!=` überladen. Ein Beispiel dazu finden Sie in »Überladen von Operatoren« auf Seite 147.

Die ToString-Methode

Die Methode `ToString` liefert die Standardtextdarstellung einer Typinstanz zurück. Die Methode wird von allen eingebauten Typen überschrieben. Hier sehen Sie ein Beispiel für die Verwendung der Methode `ToString` des Typs `int`:

```
string s1 = 1.ToString();      // s1 ist "1"
string s2 = true.ToString();   // s2 ist "True"
```

Sie können die Methode `ToString` für selbst definierte Typen wie folgt überschreiben:

```
public override string ToString() => "Foo";
```

Structs

Ein *Struct* ähnelt einer Klasse, es gibt aber auch entscheidende Unterschiede:

- Ein Struct ist ein Werttyp, während eine Klasse ein Referenztyp ist.
- Ein Struct unterstützt keine Vererbung (außer dass es implizit von `object` oder genauer `System.ValueType` abgeleitet ist).

Ein Struct kann alle Member nutzen, die eine Klasse haben kann, mit Ausnahme von parameterlosen Konstruktoren, Finalizern und virtuellen Membern.