Лекция 4. Типы данных в языке C#

Каждая сущность в программе C# является объектом, находящимся либо в стеке, либо в управляемой куче. Каждый метод определён в объявлении класса или структуры. Здесь нет свободных функций, которые определены вне контекста объявления `class` или `struct`, как в C++. Даже встроенные типы значений, такие как `int`, `long`, `long double` и т.д., имеют методы, ассоциированные с ними неявно. Фактически ключевые слова встроенных типов в C# на самом деле отображаются на типы из пространства имён `System`, представляющие их.

В управляемом мире CLR существуют два вида типов:

- *типы значений* (*value types*) — определяются в C# с применением ключевого слова `struct`. Экземпляры типов значений — единственный вид экземпляров, которые могут располагаться в стеке. Эти типы подобны структурам C++ в том смысле, что по умолчанию они копируются по значению при передаче в виде параметров методам или присваивании другим переменным.
- *ссылочные типы* (*reference types*) — часто определяются в C# с применением ключевого слова `class`. Они называются ссылочными типами потому, что переменные, используемые для манипуляций ими, в действительности являются ссылками на объекты из управляемой кучи. Фактически переменные ссылочных типов подобны переменным типов значений, которые имеют ассоциированный тип и содержат указатель на объект в куче. Для определения таких объектов, создаваемых в куче, используется определение `class`. Программисты C++ могут воспринимать переменные ссылочных типов как указатели, которые не нужно разыменовывать для доступа к объектам.

Часто под ссылочными типами в C# подразумеваются объекты, находящиеся в управляемой куче. Однако для взаимодействия с объектами из управляемой кучи используется переменная ссылочного типа. Переменные ссылочных типов имеют тип, ассоциированный с ними, и содержат указатель на объект, на который ссылаются. Например, тип ссылочной переменной может совпадать с классом, на который она указывает, с базовым типом объекта либо типом интерфейса, который данный объект реализует. Естественно, несколько переменных ссылочных типов могут указывать на один и тот же объект в одно и то же время. В отличие от них, типы значений содержат в себе данные, а не указатели на данные.

Значимые типы

В .NET Framework простейшие типы, в основном числовые и логический, являются значимыми.

Значимые типы (*value types*) — это переменные, содержащие сами данные, а не ссылку на них. Экземпляры значимых типов хранятся в области памяти под названием *стек* (*stack*), в которой исполняющая среда создаёт, считывает, обновляет и удаляет их быстро и с минимальной затратой ресурсов.

Существует три основных группы значимых типов:

- встроенные типы;
- структуры;
- перечислимые типы.

Все эти типы являются потомками базового типа `System.ValueType`.

Встроенные типы

Встроенные типы — это базовые типы, поставляемые с .NET Framework, на их основе строятся все остальные типы. Все встроенные числовые типы являются значимыми. Числовой тип следует выбирать, исходя из размера его значения и желаемой точности вычислений. В табл. 4.1 перечислены наиболее востребованные встроенные типы.

Эти типы используются настолько часто, что в C# для них определены *псевдонимы*. Псевдоним — это сокращённый эквивалент полного имени типа; большинство программистов предпочитает использовать более короткие псевдонимы.

Исполняющая среда оптимизирует производительность 32-разрядных целочисленных типов (`int` и `uint`), поэтому их следует использовать для счётчиков и других часто используемых переменных. Для операций с плавающей запятой лучше всего подходит тип `double`, поскольку такие операции оптимизированы аппаратно.

Табл. 4.1. Встроенные значимые типы

Тип C#	Полное название	Размер в байтах	Возможные значения
	Беззнаковые целые числа		
byte	System.BSByte	1	0 — 255
ushort	System.UInt16	2	0 — 65 535
uint	System.UInt32	4	0 — 4 294 967 295
ulong	System.UInt64	8	0 — 18 446 744 073 709 551 615
	Знаковые целые числа		
sbyte	System.SByte	1	–128 — 127
short	System.Int16	2	–32 768 — 32 767
int	System.Int32	4	–2 147 483 648 — 2 147 483 647
long	System.Int64	8	–9 223 372 036 854 775 808 — 9 223 372 036 854 775 807
	Вещественные числа		
float	System.Single	4	-3,402823E38 — 3,402823E38 (точность 7 знаков)
double	System.Double	8	–1.79769313486232E308 — 1.79769313486232E308 (точность 15—16 знаков)
decimal	System.Decimal	16	–79 228 162 514 264 337 593 543 950 335 — 79 228 162 514 264 337 593 543 950 335 (точность 28—29 значимых цифр)
	Другие значимые типы		
bool	System.Boolean	1	Логические значения true и false
char	System.Char	2	Одиночные символы Unicode
—	System.DateTime	зависит от значения	Дата и время

В .NET Framework существует более 300 значимых типов, но обычно удается обойтись приведёнными выше. При присваивании переменной значения другой переменной данные копируются из второй переменной в первую, в стеке остаются две копии этих данных.

Хотя значимые типы часто представляют простые значения, они все равно функционируют как объекты, то есть поддерживают методы. В действительности, для вывода значения в виде текста обычно используется метод `ToString` значимого типа. Обычно для этих целей переопределяется метод `ToString` базового типа `System.Object`. В .NET Framework все типы порождаются от типа `System.Object`. Такое родство позволяет создать общую систему типов, используемую в .NET Framework.

Чтобы использовать тип, сначала нужно объявить имя экземпляра этого типа. Значимые типы поддерживают неявные конструкторы, так что при объявлении такого типа его экземпляр создаётся автоматически, без употребления ключевого слова `new` (в отличие от классов, где оно необходимо). Конструктор присваивает новому экземпляру значение по умолчанию (обычно `null` или 0), но переменную следует явно инициализировать внутри объявления:

```
bool b = false;
int a = 35;
```

Структуры

Структура (*structure*) — это тип значения, который обычно используется для инкапсуляции небольших групп связанных переменных. Подобно другим значимым типам, экземпляры пользовательских типов хранятся в стеке и содержат данные, а не ссылки на них. В остальном структуры почти полностью аналогичны классам.

Для создания структур используется ключевое слово `struct`. Структура подходит для создания несложных объектов. Примерами стандартных структур являются `Point`, `Rectangle` и `Color`. Хотя точку удобно представить в виде класса, в некоторых сценариях структура может оказаться более эффективной. Например, при объявлении массива из 1000 объектов `Point` потребуется выделить дополнительную память для хранения ссылок на все эти объекты, и структура в таком случае будет более экономичным решением:

```
public struct CoOrds
{
    public int x, y;

    public CoOrds(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

При создании объекта структуры с помощью оператора `new` объект создаётся и вызывается соответствующий конструктор. В отличие от классов структуры можно создавать без использования оператора `new`. В таком случае вызов конструктора отсутствует, что делает выделение более эффективным. Однако поля остаются без значений и объект нельзя использовать до инициализации всех полей.

Следующий код демонстрирует инициализацию структуры с помощью конструктора по умолчанию и с помощью конструктора с параметрами.

```
CoOrds coords1 = new CoOrds();
CoOrds coords2 = new CoOrds(10, 20);

// Вывод результатов:
Console.Write("CoOrds 1: ");
Console.WriteLine("x = {0}, y = {1}", coords1.x, coords1.y); // будет выведено: x = 0, y = 0

Console.Write("CoOrds 2: ");
Console.WriteLine("x = {0}, y = {1}", coords2.x, coords2.y); // будет выведено: x = 10, y = 20
```

В следующем примере демонстрируется уникальная возможность структур. В нём создается объект `CoOrds` без использования оператора `new`.

```
CoOrds coords1;

// Инициализация:
coords1.x = 10;
coords1.y = 20;

// Вывод результатов:
Console.WriteLine("CoOrds 1: x = {0}, y = {1}", coords1.x, coords1.y); // x = 10, y = 20
```

Хотя по функциональности структуры не отличаются от классов, первые обычно более эффективны. Если тип лучше работает как значимый, а не ссылочный, следует определять структуру, а не класс. Структурные типы должны удовлетворять следующим требованиям:

- логически представлять одиночное значение;
- размер их экземпляра не должен превышать 16 байтов;
- после создания они не должны изменяться;
- не должны приводиться к ссылочным типам.

Перечислимые типы

Перечислимые типы — это наборы связанных символов с фиксированными значениями. Перечислимые используются для представления альтернативных значений, доступных разработчикам, использующих класс, в котором объявлено перечислимое. Так, следующее перечислимое хранит возможные обращения:

```
enum Titles : int { Mr, Ms, Mrs, Dr };
```

Если создать экземпляр типа `Titles`, Visual Studio покажет список возможных значений переменной этого типа. Хотя это целочисленная переменная, можно вывести одно из её связанных значений-строк, как показано в следующем примере:

```
Titles t = Titles.Dr,
Console.WriteLine("{0}.". t); // Выведет "Dr."
```

Перечисления предназначены для упрощения кодирования и улучшения читаемости кода, они позволяют использовать понятные символы вместо обычных численных значений.

Каждая константа, определенная в перечислении, должна быть определена со значением, находящимся в пределах диапазона лежащего в основе типа. Если значение константы перечисления не указано явно, принимается значение по умолчанию, равное 0 (если это первая константа в перечислении) или значению предыдущей константы, увеличенному на 1.

Все перечисления основаны на лежащем в основе целочисленном типе `int`, если не указано иначе. Лежащий в основе перечисления тип обязательно должен быть целочисленным. Если используется тип `int`, то его имя и двоеточие можно опустить:

```
enum Titles { Mr, Ms, Mrs, Dr };
```

Константы перечисления часто используются для представления битовых флагов. Чтобы сделать это явно, можно присоединить к перечислению атрибут из пространства имен `System` по имени `System.FlagsAttribute`. Атрибут сохраняется в метаданных, и на него можно сослаться во время проектирования для определения того, предназначены ли члены перечисления для использования в качестве битовых флагов.

Обратите внимание, что `System.FlagsAttribute` не изменяет поведение значений, определённых перечислением. Однако во время выполнения некоторые компоненты могут использовать метаданные, сгенерированные атрибутом, для обработки значения иным образом.

Ниже приведён пример перечисления битовых флагов:

```
[Flags]
public enum AccessFlags
{
    NoAccess = 0x0,
    ReadAccess = 0x1,
    WriteAccess = 0x2,
    ExecuteAccess = 0x4
}
```

Использовать битовые флаги можно так:

```
AccessFlags access = AccessFlags.ReadAccess | AccessFlags.WriteAccess;
Console.WriteLine(access);
```

Если скомпилировать и запустить приведённый пример, вы увидите, что метод `Enum.ToString`, неявно вызываемый из `WriteLine`, фактически выводит разделённый запятыми список всех битов, установленных в этом значении.

Ссылочные типы

В .NET Framework большинство типов — ссылочные. В стеке хранятся только адреса значений ссылочных типов — *указатели* (*pointers*). Сами значения, на которые указывает ссылка в стеке, хранятся в другой области памяти, называемой *кучей* (*heap*). Исполняющая среда автоматически управляет памятью кучи, используя процедуру под названием *сборка мусора*. Эта процедура периодически освобождает память, удаляя элементы, ссылок на которые не осталось.

Сбор мусора запускается, только когда памяти остаётся слишком мало, или вызовом метода `GC.Collect`. Автоматическое управление памятью оптимизировано для приложений, в которых большинство объектов — короткоживущие (за исключением созданных при запуске приложения). Такой подход к проектированию кода обеспечивает максимальную производительность. Сборщик мусора (`GC`) управляет всем, что касается размещения объектов. Он может перемещать объекты в любое время. При этом исполняющая среда обновляет переменные, ссылающиеся на эти объекты.

По соглашению термином *объект* обозначается экземпляр ссылочного типа, в то время как термином *значение* — экземпляр типа значений, но все экземпляры любого типа (ссылочного типа или типа значений) также унаследованы от типа `object`.

Переменные ссылочного типа инициализируются либо с помощью операции `new` для создания объекта в управляемой куче, либо присваиванием другой переменной совместимого типа. В следующем фрагменте кода две переменных указывают на один и тот же объект:

```
object o1 = new object();
object o2 = o1;
```

Исполняющая среда управляет всеми ссылками на объекты в куче. В C++ необходимо явно удалять объекты, расположенные в куче, причём в некоторый тщательно выбранный момент. Но в управляемой среде CLR за вас это делает `GC`. Это избавляет от необходимости заботиться об удалении объектов из памяти и минимизирует утечки памяти. В любой момент времени `GC` может определить количество ссылок на определённый объект в куче. Если он выясняет, что ссылок нет, значит, можно начать процесс уничтожения объекта в куче. Предыдущий фрагмент кода включает две ссылки на один и тот же объект. Первая из них, `o1`, инициализируется созданием нового экземпляра `object`, а вторая — `o2` — инициализируется присваиванием первой, `o1`. `GC` не удалит этот объект из кучи до тех пор, пока обе ссылки не выйдут из их области видимости. Если бы метод возвращал копию ссылки тому, кто его вызывает, то `GC` продолжал бы отслеживать ссылку на данный объект, даже несмотря на то, что создавший его метод уже завершился.

По умолчанию компилятор C# производит то, что называется *безопасным кодом* (*safe code*). Один из аспектов безопасности состоит в том, что программа не использует неинициализированную память. Компилятор требует, чтобы каждой переменной было присвоено значение прежде, чем можно будет работать с ней, поэтому полезно знать, как инициализируются переменные разных типов.

Значением по умолчанию для ссылок на объекты является `null`. В точке объявления можно дополнительно присваивать ссылки, полученные в результате вызова операции `new`; в противном случае они остаются установленными в `null`. Когда создаётся объект, исполняющая система инициализирует его внутренние поля. Поля, являющиеся ссылками на объекты, разумеется, инициализируются значением `null`. Поля, относящиеся к типам значений, инициализируются установкой всех битов значения в ноль. По сути, можно представлять, что все, что делает исполняющая система — это установка лежащего в основе хранилища в 0. Для ссылок на объекты это соответствует `null`-ссылке, а для типов значений — нулевому значению (или `false` для булевского типа).

Для типов значений, которые объявляются в стеке, компилятор не выполняет автоматическую инициализацию нулями. Однако перед использованием значения память должна быть инициализирована.

Преобразование типов

Необходимость в преобразовании экземпляров одного типа в другой возникает очень часто. В некоторых случаях компилятор выполняет такое преобразование неявно — когда значение одного типа присваивается переменной другого типа, и при этом не теряется точность и значение. В случае если точность может быть утеряна, требуется явное преобразование. Для ссылочных типов правила преобразования аналогичны правилам преобразования указателей в C++.

Семантика преобразования типов подобна той, которая реализована в C++. Явные преобразования в языке C# выполняются с применением синтаксиса приведения, который он унаследовал от языка C, т.е. тип, к которому нужно преобразовать, помещается в скобках перед преобразуемым типом:

```
int defaultValue = 12345678;
long value = defaultValue;
int smallerValue = (int)value;
```

В этом коде приведение (`int`) должно быть явным, поскольку размер `int` меньше, чем `long`. То есть, существует возможность того, что значение типа `long` не уместится в пространство, отведенное под `int`. Присваивание переменной `defaultValue` переменной `value` не требует приведения, поскольку `long` имеет больше места для хранения, чем `int`. Если при преобразовании теряется значение, возможно, что эта операция сгенерирует исключение во время выполнения. Общее правило гласит, что неявные преобразования гарантированно никогда не сгенерируют исключений, в то время как явные — наоборот, могут.

В языке C# неявное преобразование разрешено, если целевой тип способен хранить значения исходного типа без потери точности. Такое преобразование называется *расширяющим*. Если диапазон или точность значений исходного типа выше, чем у целевого, преобразование называется *сужающим*, и такое преобразования разрешено выполнять только явно.

При преобразовании значимого типа в ссылочный и наоборот выполняются операции упаковки и распаковки.

Упаковка преобразует значимый тип в ссылочный. В следующем примере производится упаковка значимого типа `int` в ссылочный тип `object`:

```
int i = 123;
object o = i;
```

Здесь создаётся объект `o`, базирующихся в куче, и в него копируется значение переменной `i`.

Распаковка выполняется во время присвоения значимому типу ссылочного объекта. Пример распаковки:

```
object o = 123;
int i = (int)o;
```

Здесь значение объекта `o` из кучи копируется в переменную `i`, которая находится в стеке. Распаковка требует явного приведения типов.

Упаковка и распаковка требует дополнительных операций, поэтому необходимо стараться избегать их при программировании циклических задач.

Поскольку явное преобразование может потерпеть неудачу с генерацией исключения, бывает так, что необходимо проверить тип переменной без выполнения приведения и наблюдения, получится оно или нет. В языке C# предусмотрены две операции:

- `is`;
- `as`.

Операция `is` даёт в результате булевское значение, говорящее о том, можно ли преобразовать данное выражение в указанный тип, как с помощью преобразования ссылки, так и посредством операций упаковки и распаковки. Например, рассмотрим следующий код:

```
string derivedObj = "Dummy";
object baseObj1 = new object();
object baseObj2 = derivedObj;
Console.WriteLine("baseObj2 {0} String", baseObj2 is string ? "является" : "не является");
Console.WriteLine("baseObj1 {0} String", baseObj1 is string ? "является" : "не является");
Console.WriteLine("derivedObj {0} Object", derivedObj is object ? "является" : "не является");
int j = 123;
object boxed = j;
object obj = new object();
Console.WriteLine("boxed {0} int", boxed is int ? "является" : "не является");
Console.WriteLine("obj {0} int", obj is int ? "является" : "не является");
Console.WriteLine("boxed {0} System.ValueType",
    boxed is ValueType ? "является" : "не является");
```

Вывод этого кода будет таким:

```
baseObj2 является String
baseObj1 не является String
derivedObj является Object
boxed является int
obj не является int
boxed является System.ValueType
```

Операция `is` принимает во внимание только преобразования ссылок. Это значит, что она не может проверять определённые пользователем преобразования, имеющиеся в типах.

Операция `as` подобна `is` за исключением того, что она возвращает ссылку на целевой тип. Поскольку гарантируется, что она никогда не сгенерирует исключения, то если данное преобразование невозможно, просто возвращается `null`-ссылка. Подобно `is`, операция `as` принимает во внимание только преобразования ссылок или преобразования с упаковкой/распаковкой.

Иногда требуется проверить, относится ли переменная к определенному типу, и если да, то выполнить какую-то операцию над нужным типом. Проверить переменную на принадлежность к типу можно с использованием операции `is`, а затем, если она вернет `true`, привести переменную к этому типу. Однако это будет не эффективно. Более удачный подход заключается в том, чтобы следовать идиоме применения операции `as` для получения ссылки на переменную с нужным типом, а затем проверить ее на неравенство `null`, что будет означать успешность преобразования. Таким образом, потребуется выполнить только одну операцию просмотра типа вместо двух.

Пространства имён

Язык C# поддерживает пространства имён для логического группирования компонентов. Пространства имён помогают избежать конфликтов имён между идентификаторами.

С помощью пространств имён можно определять все типы так, чтобы их идентификаторы квалифицировались пространством имен, к которому они принадлежат. Для организации компонентов можно создавать собственные пространства имён. Общая рекомендация звучит так: для наименования пространства имён верхнего уровня используйте некоторый общий идентификатор, например, название организации, а для вложенных пространств имён подойдут более специфичные идентификаторы библиотек.

Пространства имён обеспечивают отличный механизм, с помощью которого можно сделать типы более осмысленными, особенно при разработке библиотек, предназначенных для использования другими людьми. Например, можно создать общее пространство имён, такое как `MyCompany.Widgets`, и поместить в него наиболее широко применяемые типы графических элементов управления. Затем можно создать пространство имён `MyCompany.Widgets.Advanced`, куда поместить менее часто используемые, но более сложные типы. Разумеется, все они могут быть помещены в одно и то же пространство имён. Однако пользователи могут запутаться, просматривая типы и обнаруживая, что редко применяемые типы перемешаны с часто используемыми.

При выборе названия для пространства имён общая рекомендация состоит в том, чтобы название следовало форме `ИмяКомпании.Технология.*`, т.е., чтобы первые две части имени, разделённые точкой, начинались с названий компании и технологии. Тогда появится возможность развивать классификацию пространств имён дальше. Примеры можно найти в .NET Framework — например, пространство имён `Microsoft.Win32`.

Синтаксис объявления пространства имён прост. В следующем коде демонстрируется объявление пространства имён `Acme`:

```
namespace Acme
{
    class Utility {}
}
```

Пространства имён не обязательно должны ограничиваться единственной единицей компиляции (т.е. файлом исходного кода C#). Другими словами, одно и то же объявление пространства имён может существовать во многих файлах C#. Когда все скомпилировано, набор идентификаторов, включённых в пространство

имён, является объединением всех идентификаторов в каждом из объявлений этого пространства имён. Фактически это объединение пересекает границы сборок. Если несколько сборок содержат типы, определённые в одном и том же пространстве имён, то общее пространство имён состоит из всех идентификаторов во всех сборках, определяющих эти типы.

Объявления пространств имён можно вкладывать друг в друга. Это можно делать одним из двух способов. Первый способ очевиден:

```
namespace Acme
{
    namespace Utilities
    {
        class SomeUtility {}
    }
}
```

В таком случае для доступа к классу `SomeUtility` должно использоваться его полное имя, т.е. `Acme.Utilities.SomeUtility`. Ниже показан альтернативный способ определения вложенных пространств имён:

```
namespace Acme
{
}
namespace Acme.Utilities
{
    class SomeUtility {}
}
```

Эффект от этого кода будет точно таким же, как от предыдущего. Фактически первое пустое объявление пространства имён `Acme` можно опустить. Оно оставлено только для демонстрационных целей, чтобы показать, что объявление пространства `Utilities` не является физически вложенным в объявление пространства имён `Acme`.

Любые типы, которые объявляются вне пространства имён, становятся частью глобального пространства имён. Определения типов в глобальном пространстве имён всегда следует избегать. Такая практика известна как «засорение глобального пространства имён» и считается дурным тоном в программировании. Это должно быть очевидным, поскольку в данном случае нет возможности защитить типы, определённые в разных местах, от потенциальных конфликтов имён.

Рассмотрим некоторый код, в котором используется класс `SomeUtility`:

```
Acme.Utilities.SomeUtility util = new Acme.Utilities.SomeUtility();
```

Применение всегда полностью квалифицированных имён довольно громоздко. Директива `using` пространств имён позволяет избежать этого. Она сообщает компилятору, что используется всё пространство имён единицы компиляции или другое пространство имён. Ключевое слово `using` эффективно импортирует все имена из заданного пространства имён в окружающее пространство имён, которым может быть глобальное пространство данной единицы компиляции. Это демонстрируется в следующем примере:

```
using Acme.Utilities;
...
SomeUtility util = new SomeUtility();
```

Теперь код намного проще и легче читать. Директива `using`, находясь на уровне глобального пространства имён, импортирует имена типов из `Acme.Utilities` в глобальное пространство. Иногда при импорте нескольких пространств имён могут возникать конфликты имён в пространствах, содержащих одноимённые типы. В таком случае можно импортировать индивидуальные типы из пространства, создавая псевдонимы имён. Такая техника доступна благодаря средству псевдонимов C#. Можно создать псевдоним только одного класса:

```
namespace Acme.Utilities
{
    class AnotherUtility() {}
}

using SomeUtility = Acme.Utilities.SomeUtility;
...
SomeUtility util = new SomeUtility ();
Acme.Utilities.AnotherUtility = new Acme.Utilities.AnotherUtility ();
```

В этом коде идентификатор `SomeUtility` является псевдонимом `Acme.Utilities.SomeUtility`. Чтобы проиллюстрировать идею, пространство имён `Acme.Utilities` было пополнено новым классом по имени `AnotherUtility`. Этот класс должен быть полностью квалифицирован при обращении к нему, поскольку для него никакого псевдонима не объявлено. Кстати, совершенно допустимо дать псевдониму другое имя, отличное от `SomeUtility`. Хотя назначение псевдониму другого имени может быть полезно для предотвращения конфликтов имён, всё же лучше использовать псевдоним, совпадающий с исходным именем класса, чтобы избежать в будущем путаницы при сопровождении.

Если вы следуете чётким принципам разделения при определении пространств имён, то не столкнётесь с этой проблемой. В проектировании считается дурным тоном создавать пространства имён, которые содержат множество разнообразных типов, относящихся к различным группам функциональности. Вместо этого должны создаваться пространства имён с интуитивно взаимосвязанными типами внутри; это облегчит для разработчиков нахождение нужных типов. В .NET Framework неоднократно встречаются пространства имён с некоторыми общими типами, включенными в них, и с более «развитыми» типами, включёнными во вложенное пространство имён под названием `Advanced`, например, `System.Xml.Serialization.Advanced`. Во многих отношениях при создании библиотек такие рекомендации отражают принцип открытости, применяемый при создании интуитивно понятных пользовательских интерфейсов. Другими словами, выбирайте интуитивно понятные имена и группы для типов и делайте их легко обнаруживаемыми.