

DATA TYPES

By Matt Savona



Locating the Illustrious Data Type

Strongly Typed	Weak Typed
C++	PHP
C#	Python
Java	Javascript

- What is the difference? Where will you explicitly find data types?

Primitive Types

- Java and C# provide a core 8 primitive types, C# offers a few additional ones as well. How can you tell if a variable is a primitive or non-primitive type just by looking at its type?
- The core 8 primitive types you should know are:

Primitive	Size
byte	8 bit
short	16 bit
int	32 bit
long	64 bit
float	32 bit, single precision
double	64 bit, double precision
char	16 bit Unicode
boolean (bool in C#)	8 bit

- C# and Java 1.5+ offer on-the-fly boxing and un-boxing. This is awesome.

Abstract Data Types

- When dealing with object oriented languages, Abstract Data Types are merely classes designed to represent a data structure.
- Most languages offer existing implementations of popular ADTs so you don't have to write them yourself.
- They are very common, and you are probably familiar with them...
 - Linked Lists
 - Stacks
 - Queues
 - Trees
 - Associative Arrays
 - HashMaps/Dictionaryes
 - Strings

User-Defined Object Data Types

- Any class that you write is an data type you created yourself! It is common (and good) practice that all of your classes begin with a capital letter. Remember, only primitives are lower-case.
- In object oriented programming, every user-defined data type extends the `Object` class, like it or not.
- Once you begin to extend your own classes, remember that although Java treats all methods as virtual, in C# you must explicitly declare `virtual` and `override` keywords in your method signatures.
- Once you instantiate your class, the resulting object is treated is as the type of your class (obviously). You can perform explicit type casting to reduce your classes (i.e. `CAnimal Animal = (CAnimal) CDog;`), or...
- You can include methods or overrides within your class to convert it to other primitive or Object data types. Think `ToString()/toString();!`

```
MyPersonClass PersonObject = new MyPersonClass();
```

Arrays, Collections and Generics

- Arrays are structures that usually store data of the same type, and whose elements are accessed by an integer index. It is possible for arrays to be jagged and/or multi-dimensional in C#.
- Jagged arrays are “arrays of arrays”, where each element of the array can be of its own size. (`int[][] JaggedArray = new int[5][];`)
- Multi-dimensional arrays are defined by the number of dimensions which they represent but are not “arrays of arrays” - the size and dimensions of the array remain the same unless a new array is constructed. (`int[,] MDArray = new int[5,2];`)

Arrays, Collections and Generics

- Collections are just that: collections of objects. Recall the popular ADTs, like Linked Lists, Stacks and Queues (and many others, including user-defined types).
- Collections introduce a lack of type safety to your program. The purpose of a strongly typed language is to eliminate inconsistencies and potential run-time errors by accidental casting of one type into another.
- Due to the nature of collections, all elements of the collection are treated as the `Object` type, this is bad – because when you pull elements from that collection you have no guarantee of the type it was going in, you may cast it improperly later...run-time errors...bad.



Arrays, Collections and Generics

- In C# and Java 1.5+ Generics solve this issue.
- Generics permit the programmer to define the data type of the entire collection. This ensures type consistency of all elements of that collection – you always know what the object's data type as it enters and leaves the collection.

```
List<CPerson> PersonList = new List<CPerson>();
```

Vs.

```
List PersonList = new List();
```

Implicit vs. Explicit Type Conversion

- In languages where you find data types (strongly typed languages, that is) you will find varying behavior for type conversion.
- Some languages, like C++, support implicit type conversion and some people will argue that is not truly a strong-typed language for this reason.

```
short x = 5000;  
int y;  
y = x;
```

- Languages like Java and C# support implicit conversions but only in one direction. Narrowing conversions are not permitted by implicit casting due to obvious loss of precision.

Valid:

```
int x = 5;  
float y;  
y = x;
```

Invalid:

```
float x = 5;  
int y;  
y = x;
```

Implicit vs. Explicit Type Conversion

- Explicit type conversion requires the programmer to state what cast is being made.

```
float x = 5;  
int y;  
y = (int)x;
```

- C# permits the custom definition of both implicit and explicit operators (this is really cool):

```
public static [explicit|implicit] operator DataTypeToConvertTo(int x) {  
    // Do your conversion here.  
    // return ConversionResults; // x is now DataTypeToConvertTo  
}
```