Programming Languages are Not the Same

Dr.Haitham A. El-Ghareeb

Information Systems Department
Faculty of Computers and Information Sciences
Mansoura University

helghareeb@gmail.com

September 23, 2012

Programming Languages

Programming Languages

- used for controlling the behavior of a machine (often a computer).

Programming Languages

- used for controlling the behavior of a machine (often a computer).
- programming languages conform to rules for syntax and semantics.

Programming Languages

- used for controlling the behavior of a machine (often a computer).
- programming languages conform to rules for syntax and semantics.
- There are thousands of programming languages and new ones are created every year.

Programming Languages

- used for controlling the behavior of a machine (often a computer).
- programming languages conform to rules for syntax and semantics.
- There are thousands of programming languages and new ones are created every year.
- Few languages ever become sufficiently popular that they are used by more than a few people

Programming Languages

- used for controlling the behavior of a machine (often a computer).
- programming languages conform to rules for syntax and semantics.
- There are thousands of programming languages and new ones are created every year.
- Few languages ever become sufficiently popular that they are used by more than a few people
- professional programmers may use dozens of languages in a career.

Basis for Comparing Programming Languages

1. Object Orientation
2. Static vs. Dynamic Typing
3. Generic Classes
4. Inheritence
5. Feature Renaming
6. Method Overloading
7. Operator Overloading
8. Higher Order Functions & Lexical Closures
9. Garbage Collection

Basis for Comparing Programming Languages (Cont.)

10. Uniform Access
11. Class Variables/Methods
12. Reflection
13. Access Control
14. Design by Contract
15. Multithreading
16. Regular Expressions
17. Pointer Arithmetic
18. Language Integration
19. Built-In Security

Object Orientation

There are several qualities for Object Oriented Languages:

Object Orientation

There are several qualities for Object Oriented Languages:

- Encapsulation / Information Hiding

Object Orientation

There are several qualities for Object Oriented Languages:

- Encapsulation / Information Hiding
- Inheritance

Object Orientation

There are several qualities for Object Oriented Languages:

- Encapsulation / Information Hiding
- Inheritance
- Polymorphism/Dynamic Binding

Object Orientation

There are several qualities for Object Oriented Languages:

- Encapsulation / Information Hiding
- Inheritance
- Polymorphism/Dynamic Binding
- All pre-defined types are Objects

Object Orientation

There are several qualities for Object Oriented Languages:

- Encapsulation / Information Hiding
- Inheritance
- Polymorphism/Dynamic Binding
- All pre-defined types are Objects
- All operations performed by sending messages to Objects

Object Orientation

There are several qualities for Object Oriented Languages:

- Encapsulation / Information Hiding
- Inheritance
- Polymorphism/Dynamic Binding
- All pre-defined types are Objects
- All operations performed by sending messages to Objects
- All user-defined types are Objects

Static vs. Dynamic Typing

Static vs. Dynamic Typing

- The debate between static and dynamic typing has raged in Object-Oriented circles for many years with no clear conclusion.

Static vs. Dynamic Typing

- The debate between static and dynamic typing has raged in Object-Oriented circles for many years with no clear conclusion.
- Proponents of dynamic typing contend that it is more flexible and allows for increased productivity.

Static vs. Dynamic Typing

- The debate between static and dynamic typing has raged in Object-Oriented circles for many years with no clear conclusion.
- Proponents of dynamic typing contend that it is more flexible and allows for increased productivity.
- Those who prefer static typing argue that it enforces safer, more reliable code, and increases efficiency of the resulting product.

Static vs. Dynamic Typing

- The debate between static and dynamic typing has raged in Object-Oriented circles for many years with no clear conclusion.
- Proponents of dynamic typing contend that it is more flexible and allows for increased productivity.
- Those who prefer static typing argue that it enforces safer, more reliable code, and increases efficiency of the resulting product.
- A dynamic type system doesnt require variables to be declared as a spe- cific type. Any variable can contain any value or object.

Static vs. Dynamic Typing

- The debate between static and dynamic typing has raged in Object-Oriented circles for many years with no clear conclusion.
- Proponents of dynamic typing contend that it is more flexible and allows for increased productivity.
- Those who prefer static typing argue that it enforces safer, more reliable code, and increases efficiency of the resulting product.
- A dynamic type system doesnt require variables to be declared as a spe- cific type. Any variable can contain any value or object.
- Statically-typed languages require that all variables are declared with a specific type.

Generic Classes

Generic Classes

- refer to the ability to parameterize a class with specific data types.

Generic Classes

- refer to the ability to parameterize a class with specific data types.
- A common example is a stack class that is parameterized by the type of elements it contains.

Generic Classes

- refer to the ability to parameterize a class with specific data types.
- A common example is a stack class that is parameterized by the type of elements it contains.
- it allows statically typed languages to retain their compile-time type safety

Generic Classes

- refer to the ability to parameterize a class with specific data types.
- A common example is a stack class that is parameterized by the type of elements it contains.
- it allows statically typed languages to retain their compile-time type safety
- Dynamically typed languages do not need parameterized types in order to support generic programming

Inheritence

Inheritence

- the ability for a class or object to be defined as an extension or specialization of another class or object.

Inheritence

- the ability for a class or object to be defined as an extension or specialization of another class or object.
- Most object-oriented languages support class-based inheritance

Inheritence

- the ability for a class or object to be defined as an extension or specialization of another class or object.
- Most object-oriented languages support class-based inheritance
- others such as SELF and JavaScript support object-based inheritance

Inheritence

- the ability for a class or object to be defined as an extension or specialization of another class or object.
- Most object-oriented languages support class-based inheritance
- others such as SELF and JavaScript support object-based inheritance
- A few languages, notably Python and Ruby, support both class- and object-based inheritance, in which a class can inherit from another class and individual objects can be extended at run time

Inheritence

- the ability for a class or object to be defined as an extension or specialization of another class or object.
- Most object-oriented languages support class-based inheritance
- others such as SELF and JavaScript support object-based inheritance
- A few languages, notably Python and Ruby, support both class- and object-based inheritance, in which a class can inherit from another class and individual objects can be extended at run time
- Though there are identified and classified as many as 17 different forms of inheritance. Even so, most languages provide only a few syntactic constructs for inheritance

Inheritence

- the ability for a class or object to be defined as an extension or specialization of another class or object.
- Most object-oriented languages support class-based inheritance
- others such as SELF and JavaScript support object-based inheritance
- A few languages, notably Python and Ruby, support both class- and object-based inheritance, in which a class can inherit from another class and individual objects can be extended at run time
- Though there are identified and classified as many as 17 different forms of inheritance. Even so, most languages provide only a few syntactic constructs for inheritance
- Multiple Inheritence

Feature Renaming

Feature renaming is the ability for a class or object to rename one of its features (a term used to collectively refer to attributes and methods) that it inherited from a super class.

Method Overloading

- Method overloading (also referred to as parametric polymorphism) is the ability for a class, module, or other scope to have two or more methods with the same name.
- Calls to these methods are disambiguated by the number and/or type of arguments passed to the method at the call site.

Operator Overloading

ability for a programmer to define an operator (such as +, or *) for user-defined types.

Higher Order Functions and Lexical Closures

Higher Order Functions:

- functions that can be treated as if they were data objects.
  - they can be bound to variables (including the ability to be stored in collections)
  - can be passed to other functions as parameters
  - can be returned as the result of other functions

Lexical Closure: bundling up the lexical (static) scope surrounding the function with the function itself, so that the function carries its surrounding environment around with it wherever it may be used.

Garbage Collection

mechanism allowing a language implementation to free memory of unused objects on behalf of the programmer

- Reference Counting
- Mark and Sweep
- Generational

Uniform Access

All services offered by a mod- ule should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation.

Class Variables/Methods

Class variables and methods are owned by a class, and not any particular instance of a class. This means that for however many instances of a class exist at any given point in time, only one copy of each class variable/method exists and is shared by every instance of the class.

Reflection

ability for a program to determine various pieces of information about an object at run-time. Most object-oriented languages support some form of reflection. This includes the ability to determine:

- the type of the object,
- its inheritance structure,
- and the methods it contains, including the number and types of parameters and return types.
- It might also include the ability for determining the names and types of attributes of the object.

Access Control

ability for a modules implementation to remain hidden behind its public interface.

Design by Contract

Design by Contract (DBC) is the ability to incorporate important aspects of a specification into the software that is implementing it. The most important features of DBC are:

- Pre-conditions, which are conditions that must be true before a method is invoked
- Post-conditions, which are conditions guaranteed to be true after the invocation of a method
- Invariants, which are conditions guaranteed to be true at any stable point during the lifetime of an object

Multithreading

ability for a single process to process two or more tasks concurrently.

Regular Expressions

pattern matching constructs capable of recognizing the class of languages known as regular languages.

Pointer Arithmetic

ability for a language to directly manipulate memory addresses and their contents.

Language Integration

It is important for a high level language (particularly interpreted languages) to be able to integrate seamlessly with other languages.

Built-In Security

language implementations ability to determine whether or not a piece of code comes from a trusted source (such as the users hard disk), limiting the permissions of the code if it does not.

DSA

Data Structures and Algorithms

Quick Tour of C#

Brief Overview

Namespaces

```
namespace DataStructures.Professor {
class Haitham {
...
}
}
```

Classes

```
public class Haitham: Professor, IBlogger, IMozillaReMo {
public string name;
public Haitham(string name) {
this.name = name; // more fields and methods here
}
```

Attributes

```
class ProfessorHaitham {
//...
[SecretOverride("...")] public void Teach(Subjects ↩
    datastructures2012) {
//...
}
}
```

Interfaces

```
interface IMozillaReMo {
IBrowser FavouriteBrowser {get; set;}
void Browse();
}
```

Enums

```
enum CoursesITeach {
DataStructuresAndAlgorithms,
SystemAnalysisAndDesign,
ResearchMethodologies,
DataBase
}
```

Struct

```
struct Faculty {
public float name;
public Faculty(String name) {
//...
}
}
```

Methods

```
class Haitham {
//...
public void GoToLecture(string LectureName) {
Point location = this.targetDatabase.Find(targetName);
this.FlyToLocation(location);
}
}
```

Virtual and Override Methods

```
class GiantRobot {
//...
public virtual void TravelTo(Point destination) {
this.TurnTowards(destination);
while (this.Location != destination) {
this.Trudge();
}
}
}

class FlyingRobot: GiantRobot {
//...
public override void TravelTo(Point destination) {
if (this.WingState == WingState.Operational) {
this.FlyTo(destination);
// whee!
} else {
base.TravelTo(destination); // oh well...
}
}
}
```

Methods

```
class Haitham {
//...
public void GoToLecture(string LectureName) {
Point location = this.targetDatabase.Find(targetName);
this.FlyToLocation(location);
}
}
```