Programming Paradigms

CT331 Week 2 Lecture 1

Finlay Smith

finlay.smith@universityofgalway.ie

Recap

- Prog. Para.: A pattern or model of programming.
- Prog. Para.: A collection of abstract features that categorise a group of languages.

- Semantics vs syntax

- Influences:

    - Capabilities, Applications, Prog. Methods, Impl. Methods, Theoretical studies, Standardization

A language that doesn't affect the way you think about programming is not worth knowing.

Alan Perlis (Turing Award Winner, 1966)

Why learning alternative paradigms is useful

- Different paradigms make different tradeoffs.
  - What's tricky in one paradigm is "baked in" in another.
- Changes paradigms forces you to "change gears".
  - How to reverse an ordered list in Java? C? Scheme? Prolog?
  - We'll see examples of this in this module
- It may become mainstream!
  - OO was once an alternative paradigm.
- Alt. Paradigms influence mainstream
  - Garbage collection came from functional programming

Why learning alternative paradigms is useful

- It will prepare you for learning languages that we've never heard of / may not exist yet.
- Help you decide what the best tool for the job is. (compare languages)
- Help you understand languages at a deeper level.

Why learn Functional programming?

- One of the oldest paradigms. (Lisp: 1958, still widely used today!)
- Heavily based on mathematical concepts. (proofs, lamdba calc.)
- Elegant solutions. (recursion)
- Other paradigms can be interpreted in terms of Functional Programming.

Why learn Logical programming?

- Long history.
- Allows implementation of things that are difficult in other paradigms.
- VERY different.
- Helps to conceptualize logical problems.

Why learn Imperative programming?

- Earliest paradigm - as far back as punch cards and magnetic loops.
- Much closer representation of how the machine actually works.
    - Ie. “closer to the metal”.
- Can help to recognise optimisation issues / computational bottlenecks.
- Contextualizes many other systems (UNIX, LINUX, etc)

Why learn Object Oriented programming?

- Tries to represent the real world.
- Abstraction and inheritance.
- **OO is Everywhere!**

Object Oriented Languages

Java

C#

C++

VB.NET

Scala

Javascript

Python

PHP

Smalltalk

Ruby

...

Fundamentals of OO

- Everything is an object.
- Computation is performed by message passing.
- Every *object* is an *instance* of a *class* which is a grouping of similar objects.
- Inheritance described the relationships between classes.

Object Oriented programming focuses on the objects that the program represents and allows them to exhibit “behaviour”.

4 major principles of OOP

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism

Principles of OOP: Encapsulation

Data is hidden as if *encapsulated* within the object.

- Direct access to data is restricted.
- We use methods to get, set and manipulate data.
- Manipulation of data is hidden.

Object caller doesn't need to know what's actually going on behind the scenes.

We can be (fairly) sure that nobody else is fiddling with our data.

Principles of OOP: Abstraction

Functionality can be defined without actually being implemented.

- High level interfaces provide method types/names without implementation.
- Allows case specific implementation
- Allows one person to define the functionality, and another to implement.
- Allows representation to be changed without affecting “public” view of class

Helpful when designing large systems. I can write an interface for a database and be sure it returns a <String>, rather than a number, and keep writing my code without implementing it right now.

Principles of OOP: Inheritance

To gain something from somebody else...

- Classes can inherit functionality without re-implementing.
- Prevents duplication of code.

Also helpful when designing large systems. Encourages well structured code-base.

Principles of OOP: Polymorphism

Poly: many (from ancient greek)

Morphism: shape

- Objects of one class can be treated like objects of other classes.

An object of type <Bike> can be cycled.

An object of type <RacingBike> that inherits from <Bike> can also by cycled.

Two <Numbers> (say, and int and a float) can be added and subtracted as if they were the same type.