**1. PROGRAMAÇÃO ORIENTADA A OBJETOS**

Segundo Moreira Neto (2009), a programação orientada a objetos (POO) pode ser definida como um modelo que tem como base a execução de métodos (pequenas funções que atuam diretamente sobre os dados de um objeto), levando em consideração a maneira como o usuário enxerga o sistema e suas funções.

O termo Programação Orientada a Objetos foi crado por Alan Kay, criador da linguagem de programação Smalltalk-80, mais conhecida com Smalltalk. Mas, antes disso foi criado por Ole Johan Dahl e Kristen Nygaard a linguagem Simula 67, em 1967, onde já se aplicava o conceito de POO. Com o passar dos anos, foi-se descobrindo que a POO tornava a codificação mais simples de fácil manutenção, tornando assim os softwares mais duradouros, precisos e cada vez mais livres de erros. Segundo um estudo realizado por Deitel (2009), com o passar dos anos houve uma mudança significativa por parte das grandes empresas em relação ao desenvolvimento usando esse modelo.

Na programação orientada a objetos, implementa-se um conjunto de classes que definem os objetos presentes no sistema de software. Cada classe determina o comportamento (definido nos métodos) e estados possíveis (atributos) de seus objetos, assim como o relacionamento com outros objetos.

Através desse relacionamento, o programador consegue estabelecer uma relação de herança e polimorfismo entre sua classes e assim tornando o seu código mais limpo, de fácil manutenção e com menos redundâncias.

Para Moreira Neto (2009), as principais definições da orientação a objetos são:

**1.1 Classe**

“Uma classe é composta de métodos (funções) e propriedades (variáveis). Ela nada mais é que o modelo de um objeto, sendo assim e a partir dela que é criado, instanciado, um objeto.”

Abaixo temos um exemplo de classe:

|  |
| --- |
| class Pessoa {  private String nome;  public void setSobrenome(String sobrenome) { nome += “ ” + sobrenome; }  } |

**1.2 Objeto**

**“**O objeto é a representação de uma classe na memória. Sua existência só ocorre quando em tempo de execução ocorre a instanciação de uma classe, dando assim a sua origem.”

Abaixo temo um exemplo de objeto:

|  |
| --- |
| Pessoa pessoa = new Pessoa(); |

Observação: Não devemos confundir variável de referência com objeto. No exemplo acima o objeto é **Pessoa** e *pessoa* é apenas a variável que o referencia, para que assim o programador possa manipular seu estado em memória.

Sendo assim podemos concluir que o modelo conceitual de Programação Orientada a Objetos (POO), consegue expressar de uma maneira mais natural, mais próximo da realidade humana e muito mais simples a simulação/solução dos problemas humanos em um computador.

**2. História**

O termo Programação Orientada a Objetos foi crado por Alan Kay, criador da linguagem de programação Smalltalk-80, mais conhecida com Smalltalk, mas muito antes disso foi criado por Ole Johan Dahl e Kristen Nygaard a linguagem Simula 67, em 1967, onde já se aplicava o conceito de POO. Com o passar dos anos, foi-se descobrindo que a POO tornava a codificação mais simples de fácil manutenção, tornando assim os softwares mais duradouros, precisos e com cada vez menos BUGs. Segundo um estudo realizado por Deitel, autor do livro “C# Como Programar”, com o passar dos anos houve uma mudança significativa por parte das grandes empresas em relação ao desenvolvimento de software usando o modelo de programação orientada a objetos, percebemos assim uma evolução/adaptação das linguagens e até mesmo o surgimento e novas. Abaixo é listada algumas das principais linguagens de POO:

- C++ : É uma evolução do C. Em seu nome percebemos o “++” que significa que ela possui o modelo de programação estruturada do C, mais o modelo de programação orientada a objetos. É considerada uma linguagem de médio nível, pois possui características de baixo nível e alto;

- Objective C: É uma linguagem reflexiva e orientada a objetos que possui transmissão de mensagens parecidos com a do Smalltalk e do C. Hoje em dia é uma linguagem utlizada nas plataformas Mac OS e IOs da Apple;

- Java: É uma linguagem orientada a objetos, reflexiva, fortemente tipada com o gerenciamento de memória automático. Diferentemente das linguagens convencionais, que são compiladas para um código nativo, o Java é compila seus bytecodes em uma máquina virtual, gerando assim um independência de Sistema Operacional;

- C#: É uma linguagem orientada a objetos, fortemente tipada e com o gerenciamento de memória automático. Ela utiliza-se de uma plataforma chama .Net, que foi desenvolvida com um conceito parecido com o do Java, de tornar o código independente do Sistema Operacional, porém essa independência é apenas para versões do Windows. Sua sintaxe é baseada no C++ e possui fortes influências do Object Pascal e Java;

- Ruby: É uma linguagem interpretada multiparadigma, de tipagem dinâmica e forte, com o gerenciamento de memória automático. É uma linguagem usada como script e suporta programação funcional. Em Ruby tudo é tratado como objeto;

- Groovy: É uma linguagem orientada a objetos desenvolvida para a plataforma Java. Ela possui característica do Python, Ruby e Smalltalk;

**3. Principais características**

Segundo Sierra (2009), podemos citar como a principal característica na POO a herança entre as classes que permite ao código definido em uma classe ser reutilizado em outras. A partir do conceito de herança podemos extrair outros, abaixo irei listar os principais citados pela autora:

**3.1 Encapsulamento**

É a premissa de que apenas o objeto consegue acessar diretamente determinadas propriedades. Para que possa existir uma interação entre os objetos, eles devem utilizar os métodos que concedam o acesso a essa propriedades, estamos falando dos “gets e sets”. São esses métodos que permitem o acesso de agentes externos ao estado do objeto e também são uma forma de proteger o objeto. Uma propriedade sendo acessada por um método “get”, permite que caso ela não esteja formatada corretamente ocorra antes algum tipo de validação. Isso serve também para métodos “set”, onde caso o valor que queira ser atribuído à uma propriedade não esteja de acordo com a regra de negócio ele seja descartado. Assim podemos garantir uma maior integridade no estado do objeto e também garantir um maior funcionamento do software de acordo com a regra de negócios. Abaixo temos um diagrama em UML que exemplifica isso:

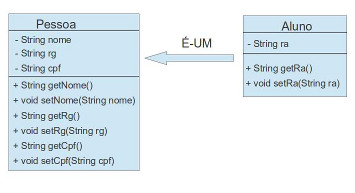


*Figura XX: representação de uma classe.*

*Fonte: Elaborado pelo autor*

**3.2 Herança**

O conceito é igual ao do mundo real, onde um ser vivo herda características de seu descendente e possui as próprias. A herança lhe permite abstrair características comuns e centralizá - las em uma superclasse, para que subclasses possam usá – las. Uma superclasse nada mais é que uma classe que centraliza métodos e propriedades comuns a mais de uma classe, e a subclasse é a classe que herda de uma superclasse e assim possui características próprias mais as da superclasse. Um bom exemplo é a relação Pessoa – Aluno, onde um Aluno antes de mais nada é uma pessoa, gerando assim uma relação de herança. Abaixo temos um exemplo dessa relação:



*Imagem criada pelo autor do trabalho de conclusão de curso.*

**3.3 Polimorfismo**

No caso do Polimorfismo eu irei citar a definição de Moreira Neto (2009), pois sua definição está mais completa:

“- Polimorfismo é o princípio pelo qual, a partir de uma subclasse, criam-se (instanciam - se) objetos utilizando referência do tipo de uma superclasse existente na hierarquia.  
 O uso do polimorfismo introduz conceitos relacionados a herança trazendo a motivação para a definição de classes abstratas, métodos abstratos, melhorando a modelagem para garantir a evolução da aplicação.”

Tendo isso como base e nos utilizando dos exemplos citados anteriormente, a relação Pessoa – Aluno, podemos extrair outros dois conceitos abordados por Kathy Sierra, o upcasting e o downcasting.

*- Upcasting:* é quando uma subclasse tornar-se uma superclasse, exemplo:

|  |
| --- |
| 1 Aluno aluno = new Aluno();  2 Pessoa pessoa = (Pessoa)aluno; //upcasting explícito  ou  2 Pessoa pessoa = aluno; //upcasting implícito |

No exemplo acima, na linha um criamos uma variável de referência denominada *aluno* que é tipo **Aluno**, e logo após instanciamos um objeto **Aluno**. Sendo assim, *aluno* aponta para um região de memória onde existe um objeto do tipo **Aluno**. Logo após, na linha dois, criamos uma variável de referência denominada *pessoa* que é do tipo **Pessoa**, em seguida atribuímos a mesma referência em memória *aluno* têm. Isso é possível pois existe uma relação de herança entre **Aluno** e **Pessoa**. Percebemos então que *aluno* conseguirá acessar todos os métodos da classe **Aluno** e **Pessoa**, porém *pessoa* não irá, mesmo que ela esteja apontando para um região de memória que possui um objeto do tipo **Aluno**, ela irá conseguir apenas acessar os métodos que seja da classe **Pessoa**, isso porque *pessoa* é uma variável de referência do tipo **Pessoa**, sendo assim ela consegue apenas acessar métodos de **Pessoa**.

- Downcasting: é quando uma superclasse torna-se uma subclasses, exemplo:

|  |
| --- |
| 1 Pessoa pessoa = new Pessoa();  2 Aluno aluno = (Aluno)pessoa; //downcasting explícito  ou  2 Aluno aluno = pessoa; //downcasting implícito |

No exemplo acima ocorre o mesmo conceito do upcasting, onde podemos alternar as referências para um objeto em memória, desde que haja uma relação de herança entre ambos.

**4. Conclusão**

Conclui-se que a programação orientada a objetos, mesmo que seja um conceito antigo, foi amplamente difundido nos anos 90 e que ainda hoje é utilizado por várias empresas. Seu modelo é considerado o mais próximo de exemplificar a realidade humana em um ambiente computacional, o que torna assim a vida de um software mais duradouro e de fácil manutenção.