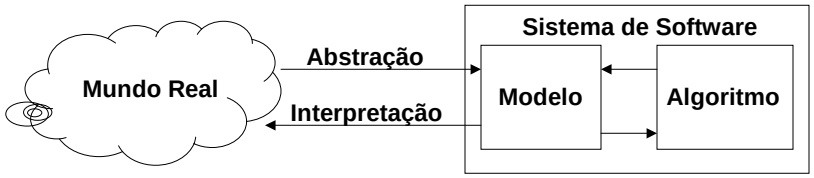
Modelo do Mundo Real



Abstração: O modelo precisa capturar apenas as características do mundo real que são importantes para o sistema de software.

Interpretação: Atribuição de significado às entidades do modelo.

Linguagens de programação utilizadas pelos programadores para descrever modelos do mundo real.

Evolução das linguagens e metodologias são uma resposta à necessidade de modelagem de sistemas cada vez mais complexos do mundo real.

Modelo do Mundo Real

Como modelamos o mundo real?

1950-1960: Foco nos algoritmos, na resolução de problemas computacionais e na busca de algoritmos eficientes. Os modelos utilizados eram modelos orientados à computação. Sistemas complexos tipicamente decompostos segundo o fluxo de controle. (Fortran, Algol)

1970-1980: Foco na informação, nos dados. Os problemas computacionais foram deixados em segundo plano. Sistemas complexos tipicamente decompostos segundo o fluxo de dados. (Cobol)

Modelo orientado a objetos: São modelos que apresentam um equilíbrio entre os dois enfoques anteriores. Estes modelos são compostos por objetos, que contém dados e realizam operações computacionais. Os sistemas complexos são decompostos em uma estrutura de objetos, classes e os relacionamentos existentes entre eles.

Princípios e Conceitos – Classes e Objetos

	Interpretação no Mundo Real	Representação no Modelo
Objeto	Qualquer coisa que possua limites bem definidos	Possui identidade única, um estado e comportamentos
Classe	Conjunto de objetos com características e comportamentos similares Os objetos são chamados instâncias da classe	Caracteriza uma estrutura de estados e comportamentos compartilhados por suas instâncias

Princípios e Conceitos – Classes e Objetos

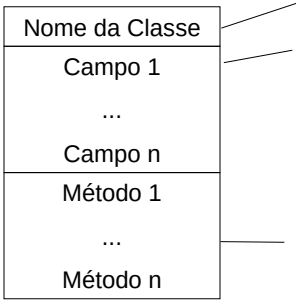
Estado: o estado de um objeto é composto por um conjunto de *campos* (ou *atributos* ou *propriedades*) e seus respectivos valores.

Comportamento: o comportamento de um objeto é definido por um conjunto de *métodos* que podem acessar ou manipular o estado.

Feature: features de um objeto refere-se ao conjunto formado pelo estado e pelo comportamento de um objeto.

Classe: uma classe é utilizada como um "template" para criação de suas instâncias (objetos). Ao invés de definir as *features* de cada um dos objetos, as classes a que os objetos pertencem tem suas features definidas.

Representação Gráfica de Classes



Indica o nome da classe descrita

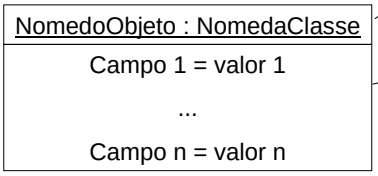
Lista de atributos da classe (opcional)

[visibilidade] [tipo] **identificador** [= valor inicial]

Lista de métodos da classe (opcional)

[visibilidade] [tipo] **identificador(**[lista de parâmetros])

Representação Gráfica de Objetos



Indica o nome do objeto e sua classe

Lista de atributos do objeto e seus valores (opcional)

[tipo] campo = valor

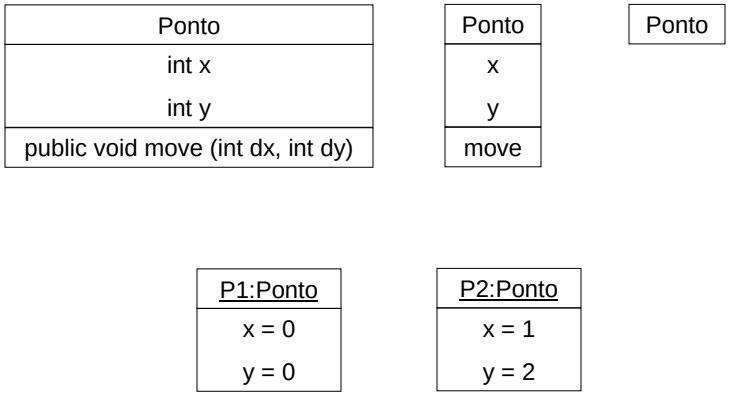
Classes e Objetos - Exemplo

```
class Ponto {
   int x, y;
   public void move (int dx, int dy){
      //...
   }
}
```

```
Ponto p1;
p1 = new Ponto();
p1.x = 0;
p1.y = 0;
```

```
Ponto p2;
p2 = new Ponto();
p2.x = 1;
p2.y = 2;
```

Classes e Objetos – Exemplo



Princípios e Conceitos – Mensagem

- Objetos comunicam-se através de mensagens.
- Mensagens podem ser interpretadas como comandos enviados aos objetos.
- O objeto receptor realiza a chamada de um de seus métodos, realizando alguma ação.
- Uma mensagem é composta pelo objeto que recebe a mensagem, o método chamado e os argumentos do método.
- A *passagem de uma mensagem* para um objeto também é conhecida por *chamada do método*.

Ex: *p1.move(10,20)*

O objeto p1 tem o seu método move chamado, e irá mover o ponto p1 10 unidades na direção x e 20 unidades na direção y.

Princípios e Conceitos – Modularidade

Um dos conceitos fundamentais da abordagem orientada a objetos é o da modularidade. Através da utilização da técnica de dividir-e-conquistar procura-se controlar a complexidade de grandes sistemas.

"Um sistema complexo deve ser dividido em módulos altamente coesos e pouco acoplados."

Coesão: as entidades pertencentes a um mesmo módulo precisam possuir alguma relação funcional.

Acoplamento: interdependência entre diferentes módulos.

Princípios e Conceitos – Modularidade

A Modularização procura:

- Módulos sejam relativamente pequenos e simples
- As interações entre os módulos sejam simples, permitindo a verificação de cada módulo de forma que se possa garantir que se cada um deles for bem comportado, o sistema todo também o será.

Os conceitos de módulo, coesão e acoplamento são todos anteriores a abordagem orientada a objetos. Na abordagem estruturada, os módulos são implementados na forma de rotinas e funções. Na abordagem orientada a objetos, os módulos são implementados em classes e pacotes (*packages*).

Princípios e Conceitos – Abstração e Encapsulamento

A abstração e o encapsulamento são ferramentas poderosas para se alcançar uma decomposição modular de um sistema.

Abstração

Separação das características essenciais e não essenciais de uma entidade.

Aproximação simples mas suficiente da entidade original.

"Os comportamentos, ou as funcionalidades de um módulo devem caracterizadas de uma forma sucinta e precisa em uma descrição conhecida como *interface contratual* do módulo. Em outras palavras, a interface contratual captura a essência do comportamento do módulo. A interface contratual é a abstração do módulo."

Princípios e Conceitos – Abstração

Módulo → Provedor de Serviço
Outros módulos → Clientes dos serviços fornecidos
Interface contratual → Contrato de Serviços

Contrato de Serviços:

- Informa quais serviços são providos, não como os serviços são realizados.
- Mesmo serviços complexos podem ter Contratos de Serviços muito simples.
- Com a garantia da execução correta do serviço, o cliente preocupa-se apenas com o contrato simples, não com o serviço complexo.

Princípios e Conceitos – Encapsulamento

O princípio de encapsulamento é complementar ao de abstração. O encapsulamento impõem que um cliente não precisa conhecer nada mais que o contrato de serviço para utilizar o serviço.

"*A implementação de um módulo deve ser separada de sua interface contratual e ocultada dos clientes do módulo.*"

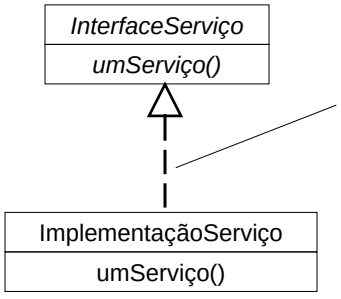
Este princípio também é conhecido por "ocultação de informação".

O encapsulamento visa a diminuição do acoplamento entre módulos.
Quanto menor o conhecimento do cliente sobre a implementação de um módulo, menor o acoplamento entre o módulo e seus clientes.
Se um cliente não conhece nada da implementação de um módulo, então sua implementação poderá ser alterada livremente.

Princípios e Conceitos – Interface

- Se a interface contratual for completamente separada da implementação, a interface pode existir sem a implementação.
- *Abstract Data Type – ADT*: Precisamos conhecer apenas sua *interface*.
- Um módulo pode ser representado por duas entidades separadas. Uma *interface* que descreve a interface contratual do módulo e uma *classe* que implementa a interface contratual.

Princípios e Conceitos – Interface (UML)



Relação do tipo
"*Implements*"

Princípios e Conceitos – Polimorfismo

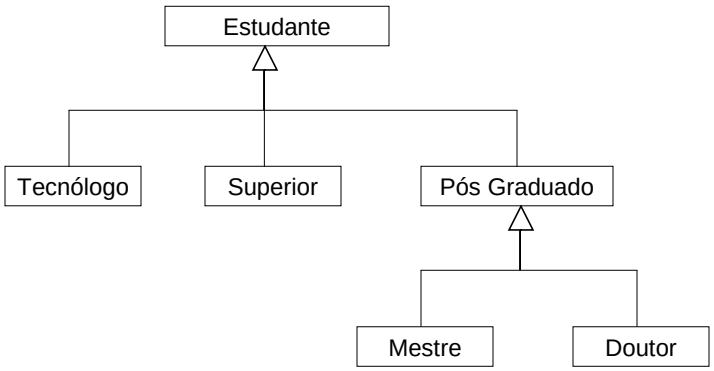
- Vários provedores de serviços podem cumprir/atender uma interface contratual.
- Estes provedores podem ser trocados sem impacto algum nos clientes.
- À esta propriedade de troca dinâmica entre diferentes módulos, sem afetar os clientes, é dada o nome de polimorfismo.
- Polimorfismo (literal): Uma entidade com múltiplas formas.
- Polimorfismo (Objetos e Classes): Interface contratual com múltiplas implementações intercambiáveis.

Princípios e Conceitos – Herança

Herança define o relacionamento entre classes.

- Quando uma classe C2 herda (*inherits*) de uma classe C1:
  - Classe C2 é uma subclasse da classe C1
  - Classe C1 é uma super classe (*superclass*) da classe C2
- Quando uma classe C2 estende (*extends*) uma classe C1:
  - Classe C2 é uma classe estendida (*extended*) da classe C1
  - Classe C1 é uma super classe (superclass) da classe C2.
- Herança define um relacionamento do tipo "*é um*", por exemplo, um elemento da classe C2 indicada acima *é um* também um elemento da classe C1, e desta forma, tudo que se aplica a C1 também se aplica a C2.

Princípios e Conceitos – Herança (UML)



Princípios e Conceitos – Níveis de Abstração

Abstração pode ser ordenada em diferentes níveis:

- Quanto mais elevado o nível de abstração, mais geral a abstração é.
- Quanto mais baixo o nível de abstração, maior a especialização da abstração.

Super classes são mais genéricas e sub classes são mais especializadas.

No exemplo, a classe Estudante representa a mais geral abstração de um estudante, sendo que cada uma de suas sub classes representam varias outras formas especializadas de estudantes.

Sobrecarga de Métodos e Construtores

Métodos e Construtores de uma classe podem ser “sobrecarregados” (*overloaded*).

“Sobrecarga é o nome dado a possibilidade de diferentes métodos e construtores de uma classe compartilharem um mesmo nome (identificador). Neste caso, o nome é sobrecarregado com múltiplas implementações.”

- As diferentes implementações devem diferir no número de parâmetros recebidos e/ou no tipo do parâmetro apenas. A esta característica chamamos assinatura (*signature*) do método ou construtor.
- O tipo de retorno ou o nome das variáveis não alteram a assinatura de um método ou construtor.

Sobrecarga de Métodos e Construtores

Método	Assinatura
String toString()	()
void move(int dx, int dy)	(int,int)
int move (int dx, int dy)	(int,int)
void paint(Graphics g)	(Graphics)

Regra da Sobrecarga:

Se dois métodos ou construtores de uma mesma classe possuírem diferentes assinaturas, então eles poderão compartilhar o mesmo nome, isto é, poderão sobrecarregar o mesmo nome.

Herança - Java

- Herança é um mecanismo de reutilização de implementações e extensão de funcionalidades de uma super classe.
- Todos os membros públicos (*public*) e protegidos (*protected*) da super classe estarão acessíveis às classes estendidas.
- Java permite apenas herança simples (não permite herança múltipla).
- A herança, em sua forma pura, ocorre entre duas classes. Entretanto há mais duas formas de herança: (1) extensão de interface e (2) implementação de interface.

[*ClassModifiers*] **class** *ClassName*
[**extends** *SuperClass*]
[**implements** *Interface1,Interface2...*]{
ClassMemberDeclarations
}

Construtores de Classes Estendidas

Processo de inicialização de uma classe estendida:

1. Inicialização dos campos herdados da super classe
2. Inicialização dos campos declarados na classe estendida

Exemplo:

```
import java.awt.Color;
public class ColoredPoint extends Point{
   public Color color;
   public ColoredPoint (double x, double y, Color color){
      super(x,y);
      this.color = color;
   }
   ...
}
```