Classes Abstratas e Interfaces

Classes Abstratas

São definidas com o propósito de criar apenas um modelo de implementação

Elas não podem ter objetos instanciados

Uso: apenas para dar uma referência genérica

- Exemplo: Classe Forma Geométrica (Módulo 3)

Polimorfismo

As classes abstratas também permitem o polimorfismo

Diversas classes podem usar referências a classes abstratas e utilizar os métodos definidos em cada uma delas

Algumas características:

- Classes abstratas podem ter tanto métodos abstratos quanto não abstratos
- Uma classe não abstrata que herda de uma classe abstrata com métodos abstratos DEVE redefinir o corpo desses métodos
- Classes não abstratas não podem ter métodos abstratos
  - Se uma classe tiver métodos abstratos, ela DEVE ser abstrata

Herança Múltipla

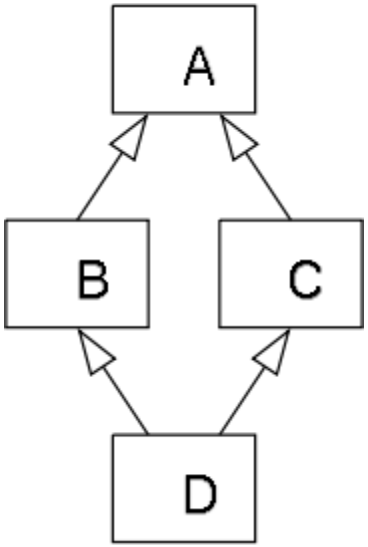
Herança múltipla é um caso específico de polimorfismo

- Uma classe possui a relação “é-um” com mais de um antecessor

Java não suporta Herança Múltipla!

- Na verdade, não é possível realizar a operação **extends** com mais de uma classe

O problema do diamante



O “problema do diamante” (às vezes chamado “deadly diamond of death”) é uma ambiguidade que surge quando duas classes B e C herdam de A, e uma classe D herda de ambas B e C. Se houver um método em A sobrescrito em B e C, mas não em D, qual versão do método D vai executar, a de B, ou de C?

Por exemplo, no contexto de desenvolvimento de software GUI, um **Button** pode herdar de ambas as classes **Rectangle** (pela aparência) e **Clickable** (para funcionalidades manipulação). As classes **Rectangle** e **Clickable** ambas herdam de **Object**. Agora, se o método **equals** é chamado para um objeto **Button** e não existe tal método nessa classe, mas existe um método sobrescrito em **Rectangle** e **Clickable**, qual deles será chamado?

Como resolver o problema anterior??

Utilizar **Interfaces**

Interface é um “contrato” no qual um objeto “compromete-se” a implementar todos os métodos ali definidos

Interface = classe abstrata somente com métodos abstratos

- Programador define interface e compilador “enxerga” uma classe abstrata com métodos abstratos

Interface NÃO DEVE ter corpo de métodos

Como realizar herança com interfaces?

Declaração de Interface

```
public interface nome_da_interface
{
        cabeçalho_do_metodo_1(parametros);
        cabeçalho_do_metodo_2(parametros);
        cabeçalho_do_metodo_3(parametros);
}
```

Implementando uma interface

```
public class nome_classe implements nome_da_interface
{
        cabeçalho_do_metodo_1(parametros){
         corpo do método 1
        }

        cabeçalho_do_metodo_2(parametros) {
         corpo do método 2
        }


}
```

Mas e a Herança Múltipla?

Uma classe filha deve estender a classe pai e implementar as demais interfaces

Neste caso é possível implementar mais de uma interface

Lembre-se: TODOS os métodos definidos nas interfaces devem ser redefinidos

Sendo assim

```
public class ClasseFilha extends ClassePai
                        implements nome_da_interface1,
                                   nome_da_interface2,...
{
  ....
}
```

O pacote java.lang

Ao começar a explorar uma classe, seja ela do próprio Java ou de algum framework, você perceberá que as classes base do Java, aquelas que integram o pacote **java.lang**, representam boa parte do código.

Você nunca vai ver o **import** de uma classe do **java.lang**, visto que elas são carregadas implicitamente em todas as classes.

O pacote **java.lang** é a base do Java. As classes desse pacote representam os fundamentos da linguagem, o ponto de partida de qualquer outra estrutura que venha a ser construída sobre ela. Isso requer bastante estudo e atenção para utilizar os elementos da base de forma eficiente e correta.

As classes base do Java foram criadas há muito tempo e vêm sendo aperfeiçoadas e refinadas desde então, sem causar qualquer impacto ao que já foi construído utilizando tais classes.

Agora a classe Object

A classe **Object** é a principal e mais básica classe do Java. Ela é a raiz da hierarquia de classes do Java, a superclasse de todas as classes, direta ou indiretamente

---

Todas as outras a têm como origem, e, portanto, herdam seus métodos, ou seja, toda classe em Java, mesmo as que não pertencem à API, estendem Object, ainda que isso não tenha sido explicitamente declarado

Sendo assim, toda instância de uma classe implementada em Java é um Object e herda os métodos declarados nesta superclasse

Como base para todas as classes, **Object** define alguns comportamentos comuns que todos objetos devem:

- **equals():** serem comparados uns com os outros
- **toString()**: poderem ser representados como texto
- **hashCode():** possuírem um número que identifica suas posições em coleções baseadas em hash

O método toString()

Representação textual de objetos

O Java usa o método toString() toda vez que for necessário converter um objeto em String, ou seja, para obter ume representação textual do objeto.

Ele é definido na classe Object, portanto é herdado por todos os objetos - todos os objetos são capazes de gerar uma representação textual.

Mas o método de Object não conhece as classes derivadas, não *sabe* como o objeto deve ser representado. Por isso usa um padrão:

- o nome da classe seguido por um '@' e pelo hashCode em hexadecimal da instância em questão

```
public String toString() {
    return getClass().getName() + "@" +
           Integer.toHexString(hashCode());
}
```

Exemplo

```
public class Pessoa {
      private final String nome;
      private int idade;
      public Pessoa(String nome, int idade) {
            this.nome = nome;
            this.idade = idade;
      }
      public static void main (String[] args) {
          Pessoa pessoa = new Pessoa("fulano", 21);
          System.out.println (pessoa);
          // equivale a System.out.println(pessoa.toString());
      }
  }
```

vai imprimir algo como "Pessoa@1b533b0" (provavelmente com outro número).

Esse formato normalmente não é o que queremos! Por isso temos que sobrescrever o método toString() para obter a representação desejado.

Exemplo

```
public class Pessoa {
        private String nome;
        private int idade;

    public Pessoa(String oNome, int aIdade) {
        nome = oNome;
        idade = aIdade;
    }
    @Override
    public String toString() {
        return nome + "(" + idade + ")";
    }
    public static void main(String[] args) {
        Pessoa pessoa = new Pessoa("fulano", 21);
        System.out.println(pessoa);
    }
}
```

Os métodos equals() e hashcode()

Definições

equals (Object obj): método fornecido por java.lang.Object que indica se algum outro objeto passado como um argumento é "igual" à instância atual. A implementação padrão fornecida pelo JDK é baseada na localização da memória - dois objetos são iguais se, e somente se, estiverem armazenados no mesmo endereço de memória.

hashcode (): método fornecido por java.lang.Object que retorna uma representação inteira do endereço de memória do objeto. Por padrão, esse método retorna um número inteiro aleatório que é exclusivo para cada instância. Esse número inteiro pode mudar entre várias execuções do aplicativo.

Quando comparamos duas variáveis de referência no Java, o == verifica se elas se referem ao mesmo endereço de memória:

```
Conta c1 = new Conta(100);
Conta c2 = new Conta(100);
if (c1 != c2) {
      System.out.println("objetos
      referenciados são diferentes!");
}
```

Exemplo

```
public class Student {
   private int id;
   private String name;
   public Student(int id, String name) {
      this.name = name;
      this.id = id;
   }
   public int getId() {
      return id;
   }
   public void setId(int id) {
      this.id = id;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
}
```

```
public class HashcodeEquals {

   public static void main(String[] args) {

      Student alex1 = new Student(1, "Alex");

      Student alex2 = new Student(1, "Alex");

      System.out.println
      ("alex1 hashcode = " + alex1.hashCode());

      System.out.println
      ("alex2 hashcode = " + alex2.hashCode());

      System.out.println
      (“verificando igualdade entre alex1 e alex2:  "
      + alex1.equals(alex2));

   }

}
```

Mas, e se fosse preciso comparar os atributos? Quais atributos deveriam ser comparados?

O método equals recebe um Object como argumento e deve verificar se ele mesmo é igual ao Object recebido para retornar um boolean.

Se você não sobrescrever esse método, o comportamento herdado visto no exemplo é mostrado a seguir:

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

Sobrescrita do Método equals()

1. Verifique se a referência para o objeto corrente (this) é igual à referência do objeto passado como argumento usando o operador de igualdade
    1. Se as referências forem iguais devolva true
2. Verifique se o objeto passado como argumento é instância da classe corrente usando o operador instanceof
    1. Em caso negativo, retorne false
3. Compare os valores internos dos dois objetos usando o operador de igualdade para comparação de primitivos e o método equals para a comparação de objetos, incluindo Strings
    1. Se os valores forem iguais retorne true, caso contrário retorne false.

Exemplo

```
public class Conta {
   private double saldo;    // outros atributos...

   public Conta (double saldo) {
      this.saldo = saldo;
   }
   public boolean equals  (Object object) {
       if (this == object)
          return true;
       if (object instanceof Conta) {
           Conta outraConta = (Conta) object;
           if (this.saldo == outraConta.saldo)
                   return true;
       }
       return false;
   }
}
```