**Paradigmas de Linguagens de Programação**
**Lista 4**
**Prof. Sergio d Zorzo**

---

1. Explique a diferença entre encapsulamento e ocultação da informação

Ambos são critérios para modularização de programas.
A diferença é que o encapsulamento diz para agrupar em um mesmo módulo dados e processos sobre estes dados, enquanto a ocultação da informação diz para separar em um software as partes mais suscetíveis a mudanças das menos suscetíveis a mudanças, ao longo do ciclo de vida. Aquelas partes mais suscetíveis a mudanças devem ficar escondidas – ou ocultas – atraás das estruturas menos suscetíveis a mudanças, de forma que o software resultante é mais flexível, compreensível e rápido de desenvolver.

2. Explique a diferença entre coesão e acoplamento

Ambos dizem respeito a módulos de um sistema, apesar de que normalmente são conceitos associados a classes em orientação a objetos.
A coesão é o grau com o qual um módulo tem uma responsabilidade única e bem definida.
O acoplamento é o grau com o qual um módulo depende de outros módulos para seu funcionamento.
O ideal, para que um sistema seja flexível e compreensível, são módulos com alta coesão e baixo acoplamento.

3. Considere as duas classes Java abaixo

```
public class Carro1 {
  public double velocidade() {
    // Calcula e retorna a velocidade do carro
  }
  public Posicao posicaoAtual() {
    // Calcula e retorna a posicao atual do carro
  }
  private Posicao posicao;
}

public class Carro2 {
  public double velocidade() {
    // Calcula e retorna a velocidade do carro
  }
  public double latitudeAtual() {
    // retorna a latitude atual
  }
  public double longitudeAtual() {
    // retorna a longitude atual
  }
  private double latitude;
  private double longitude;
}
```

a. Em qual delas o princípio da ocultação da informação é melhor aplicado? Por que?

Na classe Carro1, pois os detalhes do posicionamento de um carro ficam ocultos atrás de uma estrutura "Posicao". Como os detalhes de posicionamento podem vir a mudar ao longo do ciclo de vida de um software (por exemplo, trocar latitude e longitude por uma representação esférica), estes não devem ficar expostos para os outros módulos do sistema.

b. Qual das duas tem menor acoplamento? Por que?

A classe Carro2, pois a classe Carro1 tem uma dependência com a classe Posicao, enquanto a classe Carro2 não depende de outras classes, já que seus atributos são todos primitivos.

4. Considere a seguinte classe Java

```
public class Carro {
  Posicao posicao;
  private void lePosicaoDoGPS() {
    // Lê sinal do GPS, calcula a posição e armazena o resultado no atributo
    // posicao
  }
  public Posicao getPosicaoAtual() {
    // Retorna a posicao atual do carro
  }
  public double getVelocidade() {
    // Calcula e retorna a velocidade do carro
  }
}
```

a. Como você implementaria o método getPosicaoAtual()? Que atributos seriam necessários?

Uma solução: dentro de getPosicaoAtual(), chamar o método lePosicaoDoGPS (para atualizar), e depois retornar o próprio atributo posicao. Não seriam necessários atributos adicionais.

b. Como você implementaria o método getVelocidade()? Que atributos seriam necessários?

Uma solução: criar um atributo "tempoLeitura", do tipo java.util.Date, e um atributo "velocidade", do tipo inteiro. O método lePosicaoDoGPS, antes de atualizar o atributo posicao, lê novas coordenadas e registra o tempo atual. Calcula a distância entre a posição armazenada no atributo posicao e a posição recém-obtida. Calcula a diferença entre tempoLeitura e tempo atual. Faz o cálculo de velocidade, e armazena no atributo "velocidade". Em seguida, armazena a nova posição e tempo nos atributos correspondentes. No método getVelocidade(), apenas é retornado o valor de velocidade.

5. Dê dois motivos para utilização de setters e getters, considerando os benefícios obtidos pelo princípio da ocultação da informação.

- Permite alterar a representação interna de alguma propriedade sem causar impacto no restante do software.
- Permite adicionar validação de valores nos setters, de forma que os valores armazenados sempre fiquem validados.
- Aumenta a compreensibilidade de um software, já que os setters/getters seguem um padrão que facilita a leitura.
- Aumenta a interoperabilidade com outro software, já que o formato dos setters/getters é sempre o mesmo, de forma que os mesmos podem ser utilizados de forma padronizada, sem conhecer detalhes de armazenamento.

6. Considere o seguinte método em Java, que realiza uma busca de histórico de um cliente, em um sistema e-commerce. Considere que um log consiste de uma entrada no banco de dados, contendo: <id do cliente, data/hora, ação realizada>

```
Log[] buscarLogs(Cliente c) {
   // executa consulta no banco, buscando os logs do cliente
   // retorna todos em um array
}
```

Explique qual o resultado, em termos de acoplamento e coesão, se mudarmos o método para:

```
Log[] buscarLogs(int idCliente) {
   // executa consulta no banco, buscando os logs do cliente
   // retorna todos em um array
}
```

O sistema fica menos acoplado, já que a nova implementação elimina a dependência com relação à classe cliente. Em alguns casos, porém, a referência explícita ao id do cliente deixa o design menos suscetível a mudanças. Caso seja necessário refinar a busca para incluir outros atributos, a assinatura do método teria que ser alterada. Já no primeiro caso, ela pode ficar fixa.
Em termos de coesão, não há mudanças identificáveis.

7. Quanto maior o número de dependências, maior o acoplamento de uma classe. Você concorda ou discorda? Por que?

Concordo, pois se uma classe depende de outras para funcionar, seu acoplamento é alto.

8. Considere o seguinte código Java

```
class Log {
  public int idUsuario;
  public Date data;
  public String acao;
}
class Cadastro {
  int usuarioLogado;
  public void inserirCliente() {
     // inserir cliente no banco de dados
     Log log = new Log();
     log.idUsuario = idUsuario;
     log.data = new Date(); // recupera a data/hora atuais
     log.acao = "inserirCliente";
     Logger logger = new Logger();
     logger.gravarLog(log);
  }
}
class Logger {
  public void gravarLog(Log l) {
    // gravar l no banco de dados
  }
}
```

Modifique as classes acima para melhor aplicar o princípio da ocultação da informação e melhorar acoplamento e coesão.

```
class Log {
  private int idUsuario;
  private Date data;
  private String acao;
```

```
  public int getIdUsuario() { return idUsuario; }
  public Date getData() { return data; }
  public String getAcao() { return acao; }
  public Log(int idUsuario, Date data, String acao) {
    this.idUsuario = idUsuario;
    this.data = data;
    this.acao = acao;
  }
  public static void gravarLog(int idUsuario, String acao) {
    Log log = new Log(idUsuario, new Date(), acao);
    // gravar log no banco de dados
  }
}
class Cadastro {
  int usuarioLogado;
  public void inserirCliente() {
    // inserir cliente no banco de dados
    Log.gravarLog(idUsuario, "inserirCliente");
    // Obs: note como a estrutura da classe Log agora fica oculta
    // das classes que solicitam um log
  }
}
```

9. O que é abstração de processos? E abstração de dados?

```
Abstração de processo consiste em estruturar um programa de forma
      que partes do processo sejam abstraídas e possam ser
      reutilizadas como unidades independentes. Exemplos são
      procedimentos e funções/métodos.
Abstração de dados consiste em estruturar um programa de forma
      abstrair detalhes de tipos de dados mais complexos, criando
      novos tipos de dados abstratos que podem ser reutilizados de
      forma independente. Exemplos são os registros e as classes.
```

10. Que recursos a orientação a objetos oferece para aplicar a ocultação da informação?

```
Classes (abstração de dados)
Atributos (abstração de dados)
Métodos (abstração de dados/processos)
Modificadores de acesso
Construtores
Herança
Interfaces
Polimorfismo
Pacotes
Exceções
Tipos genéricos
```

11. Considere o programa a seguir

```
public String concatenaStrings1(String a, String b) {
  a += b;
  return b;
}

public StringBuffer concatenaStrings2(StringBuffer a,StringBuffer b) {
  a.append(b);
  return b;
}

public void principal() {
  String saudacao1 = "Alo";
  StringBuffer saudacao2 = new StringBuffer("Alo");

  String x = concatenaStrings1(saudacao1, saudacao1);
  StringBuffer y = concatenaStrings2(saudacao2, saudacao2);
  System.out.println(x);
  System.out.println(y);
}
```

a. Explique como a criação de apelidos nos procedimentos concatenaStrings1 e concatenaStrings2 prejudica a legibilidade do programa.

A semântica da função diz que apenas o primeiro parâmetro real (vinculado ao parâmetro formal "a") sofre modificação, enquanto o segundo segue inalterado. Caso a e b sejam apelidos, porém, o segundo parâmetro pode ser alterado, ainda que isso não fique aparente no código.

b. Explique porque x e y possuem valores diferentes ao final da execução do procedimento principal, mesmo que os procedimentos concatenaStrings1 e concatenaStrings2 aparentemente façam a mesma coisa.

Em Java, a passagem de parâmetros é sempre por valor, ou seja, alterar um parâmetro formal via atribuição (como no caso de concatenaStrings1, utilizando a += b) não altera o valor do parâmetro real. Portanto, saudacao1 permanece inalterado.
Porém, é possível alterar o estado de um objeto através de chamada de métodos (como no caso de concatenaStrings2, utilizando a.append(b)). Neste caso, o parâmetro real saudacao2 tem seu estado alterado, e o resultado é AloAlo.
Além disso, seria impossível modificar concatenaStrings1 para alterar o estado de saudacao1, pois nesse procedimento são utilizadas Strings, que são imutáveis.

c. Cite uma desvantagem da imutabilidade das Strings em Java

Sua manipulação (concatenação, divisão, etc) sempre resulta na criação de novos objetos. Caso haja manipulação excessiva de Strings (como logging, por exemplo), isso pode causar gasto excessivo de memória e processamento.

12. Qual a saída do programa abaixo?

```
class Posicao {
  int x, y;
}
…
Posicao p = new Posicao();
System.out.println(p.x);
System.out.println(p.y);
…
```

```
0
0
```

13. Modifique o programa anterior para que uma posição nunca possa assumir valores negativos.

```
class Posicao {
  private int x, y;
  public void setX(int x) { if(x < 0) x = 0; }
  public void setY(int y) { if(y < 0) y = 0; }
  public int getX() { return x; }
  public int getY() { return y; }
}
```

14. Implemente um método que troca os valores de dois argumentos inteiros, um pelo outro.

```
class Inteiro {
  public int valor;
}
…
void troca(Inteiro i1, Inteiro i2) {
  int temp = i1.valor;
  i2.valor = i1.valor;
  i1.valor = temp;
}
```

15. Explique o que é e como deve ser usada a chamada this(...)

```
É uma chamada para outro construtor da própria classe. Deve ser
      inserida na primeira linha de um construtor diferente, não
      podendo haver instruções antes.
```

16. Explique o que é e como deve ser usada a chamada super(...)

```
É uma chamada para um construtor da superclasse. Deve ser inserida
      na primeira linha de um construtor, não podendo haver
      instruções antes.
Todo construtor deve ter uma chamada this(...) ou super(...) em
      sua primeira instrução. Caso o programador não especifique
      explicitamente, o compilador insere automaticamente uma
      chamada super() sem argumentos.
```

17. Explique porque o programa abaixo não compila. O que deve ser alterado em “Classe” para que ele compile corretamente?

```
class SuperClasse {
   public SuperClasse(String valorInicial) {
   }
}
class Classe extends SuperClasse {
}
```

Toda classe em Java deve ter um construtor. Se não for declarado pelo programador, o compilador insere automaticamente um construtor padrão vazio. Neste exemplo, o compilador inseriu um construtor Classe() {}.
Além disso, todo construtor em Java deve ter uma chamada para outro construtor da própria classe (usando this(...)) ou para um construtor da superclasse. Se o programador não declarar explicitamente, o compilador insere uma chamada super() na primeira linha do construtor. É o que ocorre neste caso.
Em resumo, o compilador modifica o código de Classe para:

```
class Classe extends SuperClasse {
  Classe() {
    super();
  }
}
```

Como SuperClasse não tem um construtor padrão, a chamada “super();” inserida pelo compilador causa erro.

18. Explique porque o programa abaixo não compila. O que deve ser alterado em “Classe” para que ele compile corretamente e funcione da mesma forma abaixo?

```
class SuperClasse {
  private String valor;
  public SuperClasse(String valorInicial) {
    this.valor = valorInicial;
  }
}
class Classe extends SuperClasse {
  public Classe(String valorInicial) {
    valorInicial += “!”;
    super(valorInicial);
  }
}
```

A chamada ao construtor da superclasse deve ser a primeira instrução de um construtor. Neste exemplo, há uma instrução, no construtor de Classe, antes da chamada a super(...). Para poder compilar, super(valorInicial) deve vir antes.
Para manter a funcionalidade acima, deve-se acrescentar “!” ao fim da String, na própria expressão, o que é permitido. Assim:

```
class Classe extends SuperClasse {
  public Classe(String valorInicial) {
    super(valorInicial + “!”);
  }
}
```

19. O que o modificador static faz quando usado com um atributo? Para que é utilizado?

Faz com que aquele atributo não seja vinculado a nenhuma instância específica, e sim à classe. Ou seja, aquele atributo é compartilhado entre todas as instâncias (e mesmo outros objetos). Para acessá-lo, deve-se utilizar uma referência à classe, e não ao objeto (ex: Classe.atributo = 3;).
É utilizado quando se deseja criar instâncias globais únicas, compartilhadas por muitos objetos. Um exemplo são constantes. Outro exemplo é o padrão Singleton.

20. O que o modificador static faz quando usado com um método? Para que é utilizado?

Faz com que o método possa ser chamado sem haver uma instância da classe. Para acessá-lo, deve-se utilizar uma referência à classe, e não ao objeto (ex: Classe.metodo(3);).
Obviamente, métodos static não podem referenciar atributos não-estáticos, já que os mesmos podem ser chamados sem haver instâncias definidas. Já o contrário é possível.
Métodos static são utilizados para manipular atributos static, seja para ocultar informação ou para realizar cálculos diversos com esse tipo de atributos.

21. O que o modificador static faz quando usado com um bloco de código? Para que é utilizado?

Faz com que aquele bloco de código seja executado uma única vez, no carregamento da classe. Esse código é chamado pela máquina virtual, e não pode ser invocado explicitamente pelo programador. O código static, assim como métodos static, não pode referenciar atributos não-estáticos.
Blocos de código static servem como construtores para atributos static. Normalmente são utilizados para inicializar atributos static que exigem cálculos mais complexos na inicialização.

22. O código abaixo compila? Por que?

```
class Qualquer {
  static int x;
  void mudar() {
    x += 2;
  }
}
```

Sim, pois é possível acessar atributos static a partir de contextos não-static.

23. O código abaixo compila? Por que?

```
class Qualquer {
  int x;
  static void mudar() {
    x += 2;
  }
}
```

Não, pois não é permitido acessar atributos não-estáticos a partir de contextos static, já que não necessariamente existe uma instância associada (e consequentemente um valor atribuído a x) na chamada ao método "mudar()"

24. Crie código Java que representa a seguinte situação:

“Uma pessoa possui nome e sobrenome, ambos do tipo String. Um livro possui autor, que é uma pessoa, título, que é uma String e número de páginas, que é um inteiro. Um autor pode ter vários livros, mas um livro tem somente um autor.”

```
class Pessoa {
  String nome, sobrenome;
}

class Livro {
  Pessoa autor;
  String titulo;
  Int numPaginas;
}
```

25. Crie código Java que representa a seguinte situação:

“Um horário possui as horas, minutos e segundos (inteiros). Uma tarefa tem um horário de início e um horário de fim, além de um título (String) e um usuário que a solicitou (String).”

```
class Horario {
  int horas, minutos, segundos;
}

class Tarefa {
  Horario inicio, fim;
  String titulo;
  String usuario;
}
```

26. Crie código Java que representa a seguinte situação:

“Uma categoria possui nome, e subcategorias, que por sua vez também são categorias com suas próprias subcategorias, e assim por diante.”

```
class Categoria {
  String nome;
  Categoria[] subcategorias;
}
```

27. Crie código java que representa a seguinte situação:

“Um arquivo possui nome e localização (Strings) e tamanho (inteiro). Um arquivo também possui anotações. Uma anotação possui um título (String) e uma data (String). Não podem existir anotações que não estejam associadas a um arquivo. Existem dois tipos de anotação: visual e textual. Anotações visuais possuem, além de título e data, uma lista de coordenadas (array de pares de inteiros). Anotações textuais possuem, além de título e data, um texto (String).”

```
class Coordenada {
  int x, y;
}
class Arquivo {
  class Anotacao {
    String titulo, data;
    Arquivo arquivo;
    private Anotacao(Arquivo arquivo) {
      this.arquivo = arquivo;
    }
  }
  class AnotacaoVisual extends Anotacao {
    List<Coordenada> coordenadas;
    private AnotacaoVisual(Arquivo arquivo) {
      super(arquivo);
    }
  }
  class AnotacaoTextual extends Anotacao {
    String texto;
    private AnotacaoTextual(Arquivo arquivo) {
      super(arquivo);
    }
  }
  String nome, local;
  int tamanho;
  List<Anotacao> anotacoes;
  public void criarAnotacaoVisual(String titulo, String data,
      List<Coordenada> coordenadas) {
    AnotacaoVisual a = new AnotacaoVisual(this);
    a.titulo = titulo;
    a.data = data;
    a.coordenadas = coordenadas;
    anotacoes.add(a);
  }
  public void criarAnotacaoTextual(String titulo, String data,
      String texto) {
    AnotacaoTextual a = new AnotacaoTextual(this);
    a.titulo = titulo;
    a.data = data;
    a.texto = texto;
    anotacoes.add(a);
  }
}
```

28. Aponte exatamente os locais onde o código abaixo não compila. Como resolver cada um deles?

```
1:   class Numero {
2:   }
3:   class Real extends Numero {
4:     double valorReal;
5:     Real(double valorReal) { this.valorReal = valorReal; }
6:   }
7:   class Inteiro extends Numero {
8:     int valorInteiro;
9:     Inteiro(int valorInteiro) { this.valorInteiro = valorInteiro; }
10:  }
11:  class Teste {
12:    Numero somarNumerosInteiros(Numero um, Numero dois) {
13:      int result = um.valorInteiro + dois.valorInteiro;
14:      return new Inteiro(result);
15:    }
16:    Numero somarNumerosReais(Numero um, Numero dois) {
17:      double result = um.valorReal + dois.valorReal;
18:      return new Real(result);
19:    }
20:    void principal() {
21:      Inteiro i1 = new Inteiro(1);
22:      Inteiro i2 = new Inteiro(2);
23:      Inteiro i3 = somarNumerosInteiros(i1, i2);
24:      Real r1 = new Real(1.0);
25:      Real r2 = new Real(2.0);
26:      Real r3 = somarNumerosReais(r1, r2);
27:    }
28:  }
```

```
Linha 13: objetos um e dois são do tipo Numero. Para corrigir,
      devem ser alterados para tipo Inteiro, ou deve ser feito um
      casting. Ex: Inteiro iUm = (Inteiro)um.
Linha 17: idem à linha 13, porém com tipo Real.
Linha 23: o tipo de retorno da função é Numero. Para corrigir,
      deve-se alterar o tipo de retorno ou fazer um casting. Ex:
      Inteiro i3 = (Inteiro)somarNumerosInteiros(i1,i2);
Linha 26: idem à linha 23, porém com tipo Real.
```

29. Modifique o código da classe "Teste" do exercício anterior, criando um novo procedimento: somarNumeros, que funciona para qualquer tipo de número, Real ou Inteiro.

```
  Inteiro somarNumerosInteiros(Inteiro um, Inteiro dois) {
    int result = um.valorInteiro + dois.valorInteiro;
    return new Inteiro(result);
  }
  Real somarNumerosReais(Real um, Real dois) {
    double result = um.valorReal + dois.valorReal;
    return new Real(result);
  }
  Numero somarNumeros(Numero um, Numero dois) {
      if(um instanceof Inteiro && dois instanceof Inteiro)
          return somarNumerosInteiros((Inteiro)um,(Inteiro)dois);
      else if(um instanceof Real && dois instanceof Real)
          return somarNumerosReais((Real)um,(Real)dois);
      else return null; // tipos incompatíveis
  }
```

30. Repita o exercício anterior, utilizando sobrescrita de métodos para obter o mesmo efeito. Utilize a seguinte classe abstrata como ponto de partida.

```
abstract class Numero {
  abstract Numero somar(Numero outro);
}
```

```
class Real extends Numero {
  double valorReal;
  Real(double valorReal) { this.valorReal = valorReal; }
  Numero somar(Numero outro) {
      if(outro instanceof Real) {
          Real rOutro = (Real)outro;
          return new Real(valorReal + rOutro.valorReal);
      }
      return null; // tipos incompatíveis
  }
}
class Inteiro extends Numero {
  int valorInteiro;
  Inteiro(int valorInteiro) { this.valorInteiro = valorInteiro; }
  Numero somar(Numero outro) {
      if(outro instanceof Inteiro) {
          Inteiro iOutro = (Inteiro)outro;
          return new Inteiro(valorInteiro + iOutro.valorInteiro);
      }
      return null; // tipos incompatíveis
  }
}
```

31. Uma pessoa tem nome, sobrenome e idade. Utilize a interface Ordenavel e o método de ordenação apresentados em aula, e faça ordenação de Pessoas em ordem decrescente de sobrenome, depois nome e depois idade. Em outras palavras, se duas pessoas tem o mesmo sobrenome, ordene pelo nome. Se possuem o mesmo sobrenome e nome, ordene pela idade.

```
class Pessoa implements Ordenavel {
    String nome, sobrenome;
    int idade;

    public Pessoa(String nome, String sobrenome, int idade) {
        this.nome = nome;
        this.sobrenome = sobrenome;
        this.idade = idade;
    }

    public Relacao compara(Ordenavel outro) {
        Pessoa outraP = (Pessoa) outro;
        if (sobrenome.compareTo(outraP.sobrenome) < 0) {
            return Relacao.MAIOR;
        } else if (sobrenome.compareTo(outraP.sobrenome) > 0) {
            return Relacao.MENOR;
        } else {
            if (nome.compareTo(outraP.nome) < 0) {
                return Relacao.MAIOR;
            } else if (nome.compareTo(outraP.nome) > 0) {
                return Relacao.MENOR;
            } else {
                if (idade < outraP.idade) {
                    return Relacao.MAIOR;
                } else if (idade > outraP.idade) {
                    return Relacao.MENOR;
                } else {
                    return Relacao.IGUAL;
                }
            }
        }
    }
}
```

32. Implemente o seguinte diagrama de classes utilizando Java, garantindo a semântica dos relacionamentos. Insira construtores, métodos set e get, além de métodos para manutenção dos relacionamentos de lista, onde julgar necessário.