2. Orientação a Objetos em Java
Métodos Subscritos

- Sempre que você tiver uma classe que herde o método de uma superclasse, poderá subscrever esse método. Exceto se o método for declarada *final*;
- **Regras**:

  - Lista de argumentos deve coincidir (caso contrário seria sobrecarga);

  - Mesmo tipo (ou subtipo) de retorno;

  - O nível de acesso não deve ser mais restitivo que o do método subscrito;

  - Não podemos subscrever métodos *final* ou *static*;

  - O método novo não deve lançar exceções verificadas novas ou mais abrangentes;

2. Orientação a Objetos em Java
Polimorfismo

```
public class Animal {

   public void comer(){ }

}
```

Código de Subscrição Inválido	Problema no código
private void comer(){ }	?
public void comer() throws Exception { }	?
public void comer(String prato){ }	?
public String comer() { }	?

2. Orientação a Objetos em Java
Polimorfismo

```
public class Animal {

   public void comer(){ }

}
```

Código de Subscrição Inválido	Problema no código
private void comer(){ }	Modificador de acesso mais restritivo
public void comer() throws Exception { }	Exceção verificada não declarada pelo método da superclasse
public void comer(String prato){ }	Lista de argumentos alterada (sobrecarga)
public String comer() { }	Tipo de retorno alterado (não é sobrecarga)

2. Orientação a Objetos em Java
Métodos Sobrecarregados

- Métodos que permitem a reutilização de um mesmo nome de um método, porém, com argumentos diferentes;
- Os métodos podem ser sobrecarregados na mesma classe ou em subclasses;
- A decisão de qual método chamar é baseada nos argumentos;

```
public class Calculadora{

    public int somar(int a, int b){
        return a+b;
    }

    public double somar(double a, double b){
        return a+b;
    }

}
```

2. Orientação a Objetos em Java
Métodos Sobrecarregados

- **Atenção**: O tipo de referência (e não o tipo do objeto) determina qual método sobrecarregado será chamado;

```
class Animal { }
class Cachorro extends Animal{ }

class Pergunta {

    public void cruzar(Animal animal) {
        System.out.println("cruzando com um animal");
    }

    public void cruzar(Cachorro cachorro) {
        System.out.println("cruzando com um cachorro");
    }

    public static void main(String[] args) {
        Animal animal = new Animal();
        Cachorro cachorro = new Cachorro();
        Animal robbStark = new Cachorro();

        Pergunta pergunta = new Pergunta();
        pergunta.cruzar(animal); //cruzando com um animal
        pergunta.cruzar(cachorro); //cruzando com um cachorro
        pergunta.cruzar(robbStark); // ??????????????
    }
}
```

2. Orientação a Objetos em Java

Métodos Sobrecarregados

- Enquanto o método **sobrescrito** a chamar é decidido no **tempo de execução**, com base no **tipo do *objeto***, no método **sobrecarregado** é decidido em **tempo de compilação**, com base no **tipo da *referência***;
- No exemplo, é chamado o método cruzar passando a ele uma **referência** a Animal a um **objeto** Cachorro. *O compilador só conhecerá Animal, não importa que em tempo de execução tenhamos passado um Cachorro*;
- Conclui-se, que: em métodos sobrecarregados, o polimorfismo não determina qual versão sobrecarregada de um método será chamada, isso só ocorre em métodos subscritos.

2. Orientação a Objetos em Java
Métodos Sobrecarregados

- Robb irá comer como um Animal ou como um Cachorro?

```
class Animal {
   public void comer() {
      System.out.println("Animal comendo");
   }
}

class Cachorro extends Animal {

   public void comer() {
      System.out.println("Cachorro comendo");
   }
}

class Pergunta {
   public static void main(String[] args) {
      Animal animal = new Animal();
      Cachorro cachorro = new Cachorro();
      Animal robb = new Cachorro();
      animal.comer(); //Animal come
      cachorro.comer();//Cachorro come
      robb.comer(); // ?????????
   }
}
```

2. Orientação a Objetos em Java

Conversão de Variáveis de Referência

- É comum usarmos tipos genéricos de variáveis de referência para apontar para objetos mais específicos;

  Animal cachorro = new Cachorro();

- Cachorro pode chamar um método não declarado em animal? **Não**.
- Que tal um ***cast***?

  Animal animal = new Animal();

  Cachorro cachorro = (Cachorro) animal;

  cachorro.latir(); **Compila!** Mas, *java.lang.ClassCastException*

- O compilador só valida se os tipos estão na mesma árvore de herança, por isso, compilou;

2. Orientação a Objetos em Java

Conversão de Variáveis de Referência

- A conversão generalizadora, por outro lado, funciona de forma implícita;

```
Cachorro cachorro = new Cachorro();
Animal animal = cachorro; //implícita
Animal animal = (Animal) cachorro; //explícita
```

- Observem outros códigos válidos

```
Animal animal = new Cachorro();
Cachorro cachorro = (Cachorro) animal;
cachorro.latir();
```

- = Animal animal = new Cachorro();
- ((Cachorro) animal).latir();

Tipos de Retorno Válidos

- Há uma diferença entre o que você pode ***declarar*** e o que você pode ***retornar***;
- Tipos de Retorno de Métodos **Sobrecarregados**:

  - Se não for um método sobrescrito, pode alterar o tipo;
- Tipos de Retornos de Métodos **Sobrescritos**:

  - Só podem alterar um tipo se for para um subtipo do tipo declarado (retornos covariantes);

2. Orientação a Objetos em Java
Implementando Interfaces

- public class PessoaImpl ***implements*** PessoaRepositorio{ … }
- **Regras** para implementações ***concretas***:

  - Deve implementar todos os métodos;

  - Deve seguir as regras de sobrecarga vista anteriormente;

  - Não declarar novas exceções;

  - Manter a assinatura e tipo de retorno;

  * Não é necessário declarar as exceções;
- Relembrando... uma classe pode implementar mais de uma interface;

2. Orientação a Objetos em Java

Tipos de Retorno Válidos

- **Retornando valores**

  1º **return null;** para métodos que retornem objetos;

  2º public String[] getEmails() { **return new String[] {“gustavomaues@gmail.com”,”...”};** } //array

  3º public int getTipo() { int a = 1; **return a;**} //primitivo

  4º public int getTipo() { float a = 1.0f; **return (int) a;**} //primitivo com conversão explícita válida;

  5º public **void** semRetorno(){ }; //sem retorno

  6º public Animal getObject() { **return new Cachorro();**} //referência a um objeto, pode ser retornado qualquer tipo ou subtipo deste que possa ser convertido implicitamente para o tipo declarado;

2. Orientação a Objetos em Java

Construtores e Instanciação

- Os objetos são construídos;
- **Você não pode criar um objeto novo sem chamar um construtor;**
- **Toda classe tem um construtor, até as abstratas!**
- Não só o construtor da classe real, mas o construtor de cada uma de suas superclasses;
- Com exceções, um construtor sempre é chamado após a palavra-chave **new**;
- É **desejável** que uma classe tenha um construtor sem argumentos;
- Construtores podem ser sobrecarregados;
- Todo construtor chama, implicitamente, **super()**;

2. Orientação a Objetos em Java

Construtores e Instanciação

- Além das regras que já conhecemos da declaração de construtores, (nome idêntico ao da classe, sem retorno, com qualquer modificador, com ou sem argumentos, etc), cabe saber:
- Se não declararmos um construtor sem argumentos, o compilador cria um padrão. Porém, se declararmos um com argumentos, o compilar não irá criar um construtor padrão;
- A primeira declaração de um construtor será, obrigatoriamente, **this()** ou **super()**; Se não colocarmos, o compilador o faz;
- **super()** pode ter argumentos para passar ao construtor da superclasse;
- O construtor padrão é apenas o que o compilador fornece. Nós, no máximo, criamos um construtor sem argumentos;

2. Orientação a Objetos em Java

Construtores e Instanciação

- Apenas métodos e variáveis *static* podem ser acessador por uma chamada super(); **Exemplo**: super(Color.*BLUE*);
- Os construtores das classes ***abstratas*** são chamados sempre que uma subclasse ***concreta*** é instanciada;
- **As interfaces não tem construtores**; pois não fazem parte da árvore de herança de um objetos;
- **Somente é possível chamar um construtor dentro de outro construtor**;
- Atenção ao uso do **this();** se você inserí-lo, o construtor não irá acrescentar automaticamento o **super();**

2. Orientação a Objetos em Java
Construtores Sobrecarregados

- Mesma lógica dos métodos. Temos vários construtores com argumentos diferentes;

```
class Animal {

    private String nome;

    public Animal() { }
    public Animal(String passandoumNome){
        this.nome = passandoumNome;
    }
}
```

- O que aconteceria se colocássemos o **this();** no construtor com argumento?
- **Experimente**!

```
class Animal {

    private String nome;

    public Animal() {
        this("Robb Stark Lobato");
    }
    public Animal(String passandoumNome){
        this.nome = passandoumNome;
    }
}
```

2. Orientação a Objetos em Java
Variáveis e Métodos *STATIC*

- **Uso em métodos:** quando queremos um comportamento *independente da instância (do estado do objeto)*;
- **Uso em variáveis:** quando queremos um valor independente da instância;
- Variáveis e Métodos estáticos não precisam de instâncias, são membros de classe.
- Mas, mesmo que hajam instâncias, elas compartilham da mesma variável, ou seja, só haverá uma cópia;
- Mesmo que você crie uma instância e acesse a variável ou método estáticos através dela, nada altera o comportamento acima;
- Métodos estáticos não podem ser sobrecarregados;
- Métodos estáticos não acessam variáveis de instância, nem métodos não estáticos;

2. Orientação a Objetos em Java
Variáveis e Métodos *STATIC*

```
public class Variaveis {

    static int variavelClasse = 0;
    int variavelinstancia = 0;

    public Variaveis() {
        variavelClasse += 1;
        variavelinstancia +=1;
    }

    public static void main(String[] args) {
        Variaveis variavel = new Variaveis();
        new Variaveis();
        new Variaveis();
        System.out.println("variavelClasse = "+Variaveis.variavelClasse);
        System.out.println("variavelinstancia = "+variavel.variavelinstancia);
    }
}
```

```
variavelClasse = 3
variavelinstancia = 1
```

2. Orientação a Objetos em Java

Acessando Variáveis e Métodos STATIC

- Para acessar um método ou variável static, basta usar o operador ponto no nome da classe;
- Exemplo:

  Cor.AZUL_MARINHO;

  Mes.DEZEMBRO;

  FacesContext.*getCurrentInstance()*;

  JSFUtil.*adicionarMensagemErro()*;

Exercício

- Crie uma interface com pelo menos 2 (dois) métodos sobrecarregados e mais um outro qualquer;
- Implemente essa interface em dois níveis de hierarquia:

  - uma classe abstrata A implementará um método sobrecarregado e o “outro qualquer”;

  - uma subclasse de A implementará o outro método sobrecarregado; e sobrescreverá o “método qualquer”;
- Na superclasse, crie um construtor com argumento;
- Na subclasse, crie os construtores necessários;
- Crie um classe com um método principal que execute variadas chamadas aos métodos. Use a criatividade;