CONCEITOS BÁSICOS DE ORIENTAÇÃO A OBJETOS

ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

Daniel Cordeiro

Escola de Artes, Ciências e Humanidades | EACH | USP

O QUE É HERANÇA?

Definição
É a capacidade de linguagens orientadas a objetos de permitir que classes herdem estados e comportamentos comuns a outras classes.

```
class MountainBike extends Bicycle {
    // novos campos e métodos que definem
    // uma mountain bike
}
```

`MountainBike` é uma subclasse de `Bicycle`.
`Bicycle` é a superclasse de `MountainBike`.

Relação é-um
Herança define uma relação é-um: `MountainBike` é uma `Bicycle`.

HERANÇA

- Classes definem a estrutura e comportamento de objetos
- Herança é a característica de linguagens OO que permite que novas classes sejam criadas usando classes pré-existentes como base
- A nova classe *herda* os campos e métodos da classe original
- Dizemos que a nova classe é uma *subclasse* da original; e a classe utilizada como base é chamada de *superclasse*.

Vantagens:

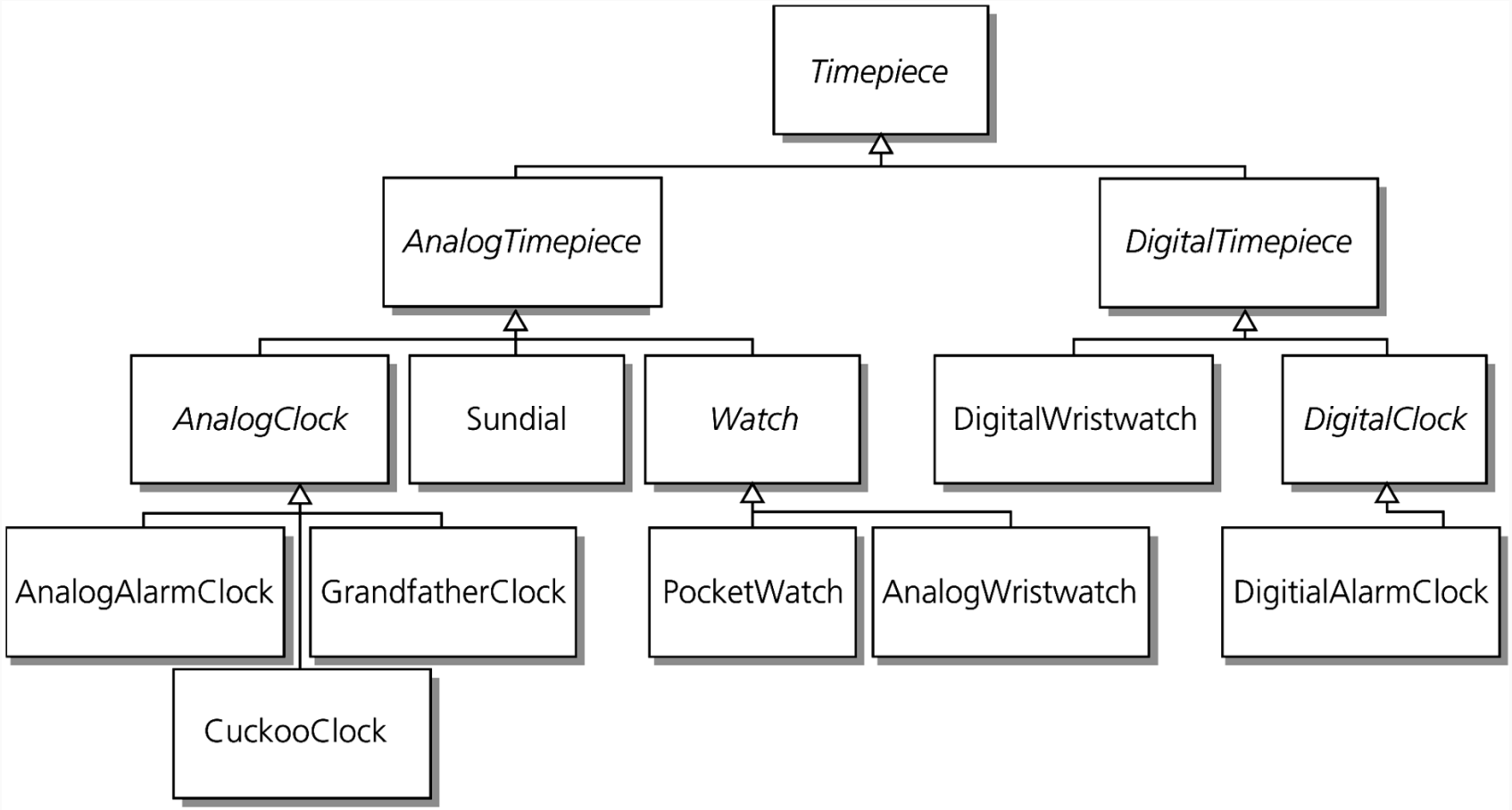
- Melhor modelagem conceitual — hierarquias de especialização são comuns na vida real
- Fatorização — herança permite que propriedades comuns sejam fatorizadas, i.e., definidas apenas uma vez
- Refinamento do projeto e validação — construção de classes com base em outras bem testadas produzirá menos defeitos
- Polimorfismo (mais sobre isso daqui a pouco)

EXEMPLO DE HERANÇA

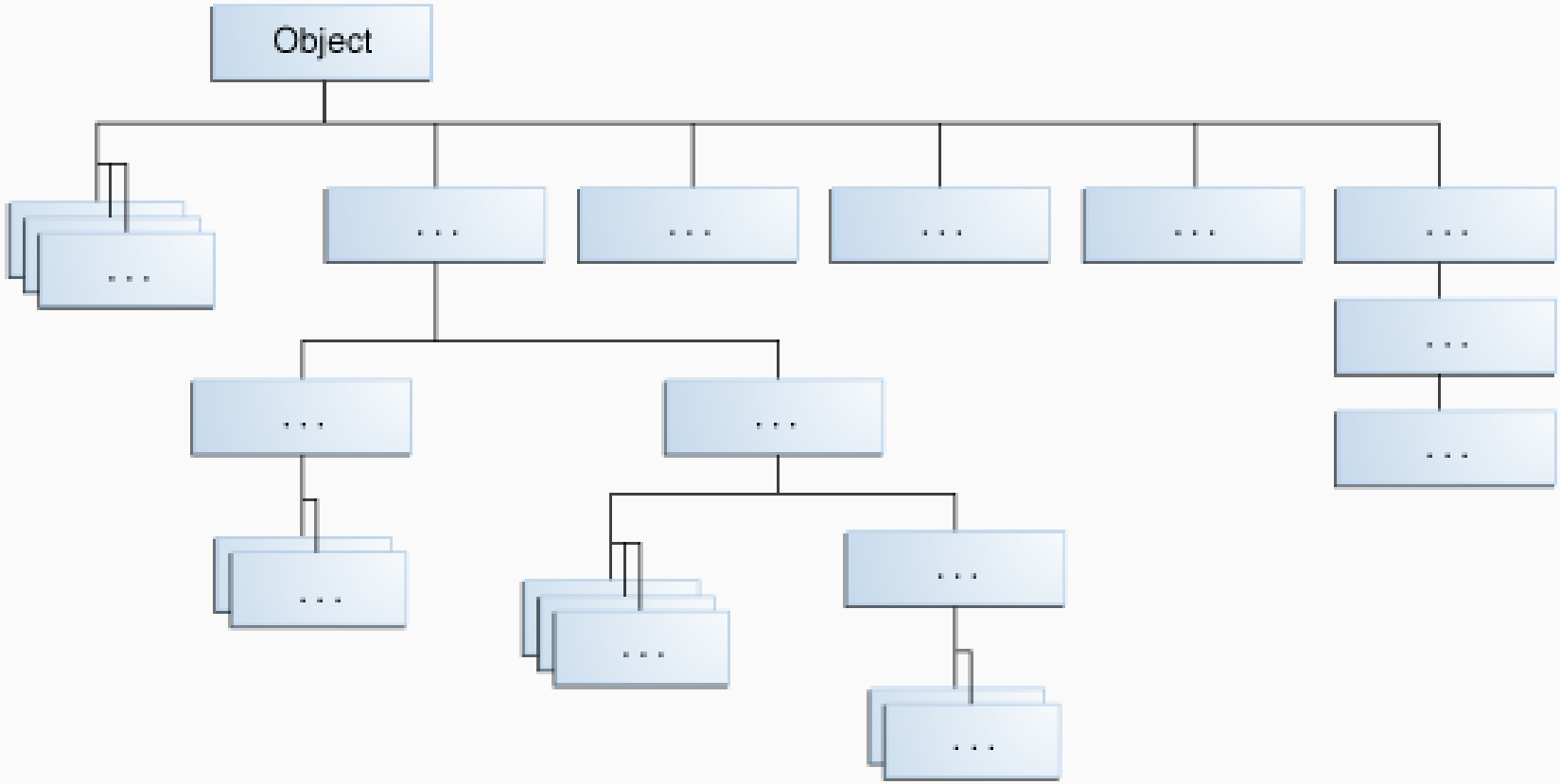
```java
public class Bicycle {

  // a classe Bicycle tem três campos
  public int cadence, gear, speed;

  // a classe Bicycle tem um construtor
  public Bicycle(int startCadence,
                 int startSpeed,
                 int startGear) {
      gear = startGear;
      cadence = startCadence;
      speed = startSpeed;
  }

  // a classe Bicycle tem quatro métodos
  public void setCadence(int newValue) {
      cadence = newValue;
  }
  public void setGear(int newValue) {
      gear = newValue;
  }
  public void applyBrake(int decrement) {
      speed -= decrement;
  }
  public void speedUp(int increment) {
      speed += increment;
  }

}
```

```java
public class MountainBike extends Bicycle {

  // a subclasse MountainBike adiciona um campo
  public int seatHeight;

  // a subclasse MountainBike tem um construtor
  public MountainBike(int startHeight,
                      int startCadence,
                      int startSpeed,
                      int startGear) {
      super(startCadence, startSpeed, startGear);
      seatHeight = startHeight;
  }

  // a subclasse MountainBike adiciona um método
  public void setHeight(int newValue) {
      seatHeight = newValue;
  }
}
```

EXEMPLO #2


Timepiece
AnalogTimepiece
DigitalTimepiece
AnalogClock
Sundial
Watch
DigitalWristwatch
DigitalClock
AnalogAlarmClock
GrandfatherClock
PocketWatch
AnalogWristwatch
DigitialAlarmClock
CuckooClock

HIERARQUIA DE CLASSES DE JAVA



HIERARQUIA DE CLASSES DE JAVA

- A raiz da hierarquia é a classe `Object`
- Todas as classes, exceto `Object`, têm apenas uma classe pai
  - A classe pai é especificada usando a palavra-chave `extends`. Ex:
    ```
    public class MountainBike extends Bicycle { ... }
    ```
- Uma classe é uma instância de todas as suas superclasses

HERANÇA DE IMPLEMENTAÇÃO

- Uma classe:
  - herda os campos e métodos visíveis de sua(s) superclasse(s)
  - pode sobrescrever (*override*) métodos para mudar seu comportamento
- Ao sobrescrever a implementação de um método, você deve obedecer o(s) contrato(s) de sua(s) superclasse(s)
  - Isso garante que a subclasse poderá ser usada em qualquer lugar que use a superclasse
  - Princípio de Substituição de Liskov (mais sobre isso numa aula futura)

Polimorfismo pode ser resumido da seguinte forma:

1. Em tempo de compilação, uma variável de um tipo mais geral pode ser usado para referenciar, em tempo de execução, um objeto de qualquer tipo mais específico
2. Em tempo de execução, o objeto referenciado por essa variável se comporta como realmente é, não como o tipo mais geral define

EXEMPLO DE POLIMORFISMO

```
abstract class Animal {
  public String nome;

  public Animal(String meuNome) { this.nome = meuNome; }
  public abstract void falar();
  public void responderChamado(String chamado) { falar(); }
}

class Cachorro extends Animal {
  public Cachorro(String meuNome) { super(meuNome); }
  public void falar() { System.out.println("Au! Au!"); }
}

class Gato extends Animal {
  public Gato(String meuNome) { super(meuNome); }
  public void falar() { System.out.println("Miau!"); }
  public void responderChamado(String chamado) {
    if(chamado.contains("comer"))
      falar();
    //else, quem pensa que é pra ficar me chamando sem motivo?
  }}

Animal toto = new Cachorro("totó");   // Cachorro É-UM Animal
Animal bichano = new Gato("bichano"); // Gato É-UM Animal
toto.responderChamado("Oi, totó!");
bichano.responderChamado("Gatinho, vem comer!");
```

MÉTODOS QUE DEVOLVEM OBJETOS

Suponha que:

```
public class Número extends Object { ... }
public class NúmeroImaginário extends Número { ... }
```

Pergunta. O método:

```
public Número devolveUmNúmero() {
    ...
}
```

1. Pode devolver um objeto do tipo Object?

MÉTODOS QUE DEVOLVEM OBJETOS

Suponha que:

```
public class Número extends Object { ... }
public class NúmeroImaginário extends Número { ... }
```

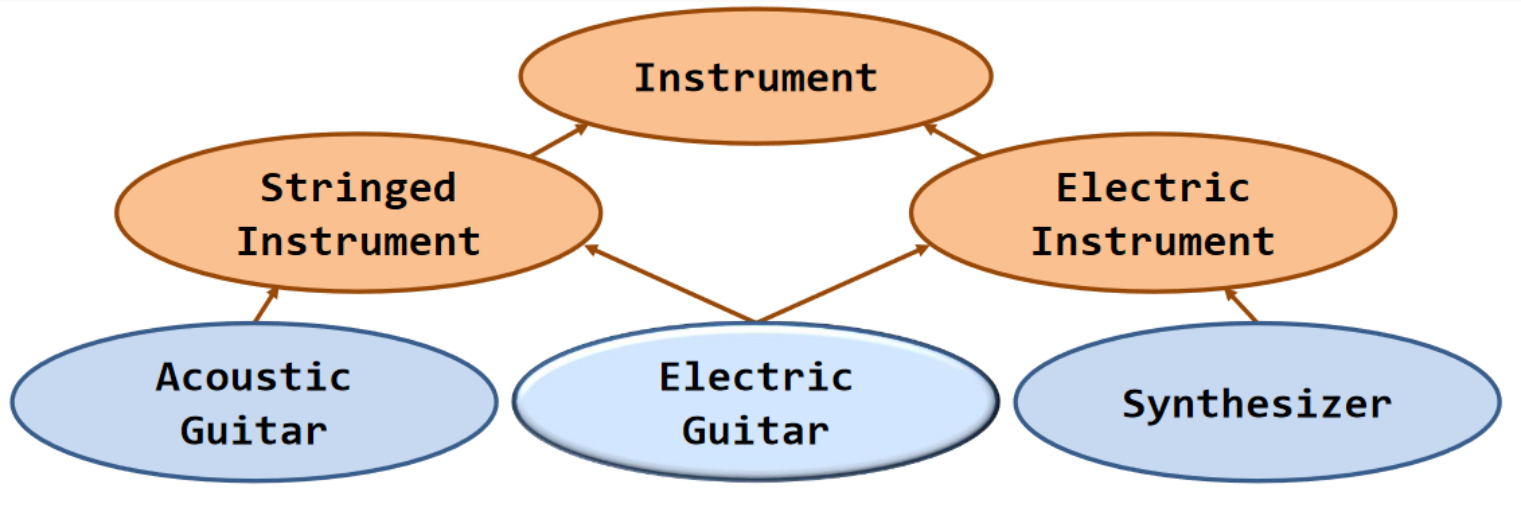
Pergunta. O método:

```
public Número devolveUmNúmero() {
    ...
}
```

1. Pode devolver um objeto do tipo Object?
2. Pode devolver um objeto do tipo NúmeroImaginário?

HERANÇA MÚLTIPLA

- Em Java, uma classe não pode estender duas classes diferentes
- Mas uma interface pode estender uma ou mais interfaces
- Uma classe pode implementar múltiplas interfaces
- Java implementa herança múltipla:
  - de tipos (uma classe pode implementar múltiplas interfaces)
  - com métodos `default` de interfaces



CONSTRUTORES E HERANÇA

```
class Animal {
   int peso;

   Animal () {
     System.out.println("Eu respiro e me mexo.");
   }

   Animal (int peso) {
     this.peso = peso;
     System.out.println("Eu respiro e me mexo e peso" + peso + " gramas");
   }
}

class Cachorro extends AnimalVertebrado {
   Cachorro () {
     System.out.println("Eu tenho coluna vertebral e lato.");
   }

   Cachorro (int peso) {
     super(peso);
     System.out.println("Eu tenho coluna vertebral e lato.");
   }

   public static void main(String args[]) {
     Animal totó = new Cachorro ();
     Cachorro rex = new Cachorro (12000);
   }
}
```

HERANÇA E SOBREPOSIÇÃO DE MÉTODOS (*METHOD OVERRIDING*)

```
class Veículo {
    void movimente () {
        System.out.println ("Eu me movimento por aí.");
    }
}

class Carro extends Veículo {
    void movimente () {
        super.movimente (); // opcional
        System.out.println("Eu gasto combustível, gero trânsito e poluo o ar.");
    }
}

class Ferrari extends Carro {
    void movimente () {
        super.movimente (); // opcional
        System.out.println("Sou vermelha, super-da-hora mas faço um barulhão.");
    }
}

class Bicicleta extends Veículo {
    void movimente () {
        super.movimente (); // opcional
        System.out.println("Faço bem para a saúde física e mental e não poluo.");
    }
}
```

SOBRECARGA DE MÉTODO (*METHOD OVERLOADING*)

```
public class DataArtist {
    ...
    public void draw(String s) {
        ...
    }
    public void draw(int i) {
        ...
    }
    public void draw(double f) {
        ...
    }
    public void draw(int i, double f) {
        ...
    }
}
```

- Os métodos são diferenciados pelo número e tipo de argumentos
- Você não pode declarar mais de um método com o mesmo nome e tipos de argumentos
- O compilador não considera o tipo do retorno para diferenciar os métodos

CLASSES E MÉTODOS ABSTRATOS

- Uma *classe abstrata* é uma classe declarada `abstract` e que pode ou não incluir métodos abstratos. Classes abstratas não podem ser instanciadas, mas podem ter subclasses
- Um *método abstrato* é um método declarado sem uma implementação
- Se uma classe inclui métodos abstratos, ela deve ser declarada como abstrata

CLASSES E MÉTODOS ABSTRATOS

```java
public abstract class Network {
    String userName;
    String password;

    Network() {}

    /**
     * Publica os dados para qualquer que seja a rede
     */
    public boolean post(String message) {
        // Autentica antes de enviar, toda rede usa seu método de autenticação
        if (logIn(this.userName, this.password)) {
            // Envia os dados da postagem
            boolean result =  sendData(message.getBytes());
            logOut();
            return result;
        }
        return false;
    }

    abstract boolean logIn(String userName, String password);
    abstract boolean sendData(byte[] data);
    abstract void logOut();
}
```

Exemplo de https://refactoring.guru/pt-br/design-patterns/template-method/java/example

CLASSES ABSTRATAS VS. INTERFACES

- Classes abstratas e interfaces são parecidas, ambas não podem ser instanciadas
- Em classes abstratas, você pode declarar campos que não são estáticos e final, e declarar métodos concretos que sejam `public`, `protected` e `private`
- Em interfaces, todos os campos são `public`, `static` e `final`

CLASSES ABSTRATAS VS. INTERFACES

Qual usar?

- Use classes abstratas se:
  - você quiser compartilhar código entre classes relacionadas
  - você espera que as classes que estenderem sua classe abstrata tenham muitos métodos e campos em comum, ou precisem de modificadores de acesso que não sejam `public`
  - você quiser declarar campos e métodos que não sejam estáticos (assim você pode definir métodos que usem o estado do objeto)
- Use interfaces se:
  - você espera que classes não relacionadas usem sua interface
  - você quer especificar o comportamento de um tipo de dado em particular, mas não se preocupa com quem vai implementar o comportamento
  - você quer se beneficiar da herança múltipla de tipo

```
public MountainBike myBike = new MountainBike();
```

- myBike é do tipo `MountainBike`
- `MountainBike` é uma classe descendente de `Bicycle` e de `Object`
- Você pode: `Object obj = new MountainBike();`
- Mas não pode: `MountainBike myBike = obj;`

Por quê?

Erro de compilação! O compilador não sabe se `obj` é mesmo do tipo `MountainBike`. Duas soluções:

1. Se você tiver certeza do tipo:
   `MountainBike myBike = (MountainBike) obj;`
2. Se não tiver certeza:

```
if (obj instanceof MountainBike) {
    MountainBike myBike = (MountainBike)obj;
}
```

BIBLIOGRAFIA

- The Java™ Tutorials:
  - https://docs.oracle.com/javase/tutorial/java/concepts/index.html
  - https://docs.oracle.com/javase/tutorial/java/IandI/index.html