

# Interfaces

Prof. Hugo de Paula



**PUC Minas**



PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS  
Curso de Engenharia de Software

# Sumário

- 1 Interfaces
- 2 Fundamentos
  - Definição de interface
  - Interfaces em Java
  - Exemplo: Timer e Wakeable
- 3 Aspectos de projeto
  - Interfaces versus Classes Abstratas
  - Métodos *default*
  - Expressões Lambda e interfaces funcionais

# Interface

“**Interface**: parte visível de um módulo a outros módulos.  
A interface deve oferecer um grupo de métodos coerente.  
Se uma interface é definida e sempre é mantida, o sistema  
ganha em extensibilidade e em baixo acoplamento.”

# Interfaces

- **Interfaces:**
  - “determinado conjunto de métodos que serão implementados em uma classe”.
  - “contrato que define tudo o que uma classe deve fazer se quiser ter um determinado status”.
- Podemos, então, especificar uma interface; e uma ou mais classes “assinariam este contrato”, comprometendo-se a implementar o que foi especificado.

# Interfaces

- Interfaces em Java possuem prioritariamente declarações de métodos (sem definição) e atributos “**public static final**”.
- A implementação fica a cargo de cada especialização desta interface.
- Interfaces são usadas para definir um protocolo de comportamento que pode ser implementado por qualquer classe na hierarquia de classes.
- Interfaces podem ser declaradas, mas não podem ser instanciadas, assim como classes abstratas.
- É uma saída elegante ao problema da herança múltipla.

# Definindo uma interface

- Definição de interfaces:
  - Declaração da interface: declara os atributos tais como nome da interface e se ela herda de outra interface.
  - Corpo da interface: contém as definições de constantes e as declarações dos métodos da interface.

```
interface nomeInterface [extends OutraInterface] {  
    corpo da Interface;  
}
```

- Para se usar uma interface usa-se a palavra-chave **implements**.

## Exemplo: Temporizador e classes acordáveis

- A classe `Timer` é um serviço que notifica objetos acordáveis que um certo tempo passou.
- Um objeto acordável deve fazer duas coisas:
  - Pedir para o `Timers` acordá-lo após certo tempo.
  - Implementar o método `wakeUp`.
- Método `letMeSleepFor` é implementado da seguinte forma:

```
public synchronized boolean letMeSleepFor(Sleeper
                                           theSleeper, long time) {
    int index = findNextSlot();
    if (index == NOROOM) {
        return false;
    } else {
        sleepers[index] = theSleeper;
        sleepFor[index] = time;
        new AlarmThread(index).start(); return true;
    }
}
```

## Exemplo: Temporizador e classes acordáveis

- Se o Timer tem espaço na lista, ele registra o dorminhoco e começa uma nova tarefa (thread) para este dorminhoco.
- Um objeto que quiser usar o Timer deve implementar o método.
- Como fazer um tipo genérico?
  - O primeiro argumento de letMeSleepFor() é um objeto do tipo Wakeable (acordável). Este tipo deve ser uma interface genérica para todos os tipos.

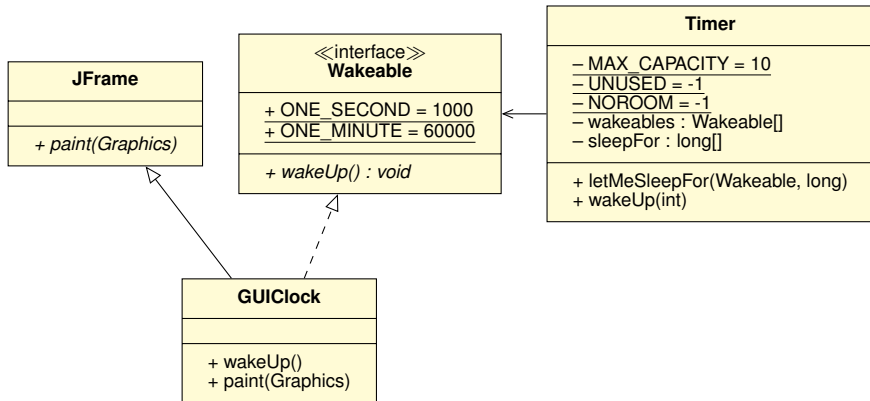
```
public interface Wakeable {  
    public void wakeUp();  
    public long ONE_SECOND = 1000;  
    public long ONE_MINUTE = 60000;  
}
```



## Exemplo: Temporizador e classes acordáveis

- A interface `Sleeper` declara o método `wakeUp()` mas não a implementa. Ela também define constantes úteis.
- As classes que implementarem esta interface “herdam” a constante e devem implementar o método `wakeUp()`.
- Todo objeto será também um dorminhoco.
- Exemplo: um relógio que deve ser acordado a cada segundo para atualizar o display do tempo.

# Exemplo: Temporizador e classes acordáveis - UML



# Porque utilizar interfaces ao invés de classes abstratas?

- Seria a classe abstrata `Sleeper` abaixo equivalente à interface?

```
abstract class Wakeable {  
    public abstract void wakeUp();  
}
```

- Resposta: Não.
- Se `Sleeper` fosse classe abstrata apenas objetos que herdassem de `Sleeper` poderiam ser utilizados no `AlarmClock`.

# Problemas com classe abstrata

- Suponha o exemplo do relógio.
- Se o relógio for um applet ele deve herdar da classe applet.
- Como Java não permite herança múltipla não seria possível tornar o relógio um dorminhoco.

# Interfaces provêm Herança Múltipla?

- Podem ser encarados como um paliativo, mas são coisas diferentes diferentes:
  - Uma classe herda apenas constantes de uma interface.
  - Uma classe não pode herdar implementações de uma interface.
  - A hierarquia de interfaces é independente da hierarquia de classes. Classes que implementam a mesma interface podem ou não estar relacionadas na hierarquia.
- Java permite herança múltipla de interfaces.

# Para que usar Interfaces?

- Use interfaces para definir protocolos de comportamento que possam ser implementados em qualquer lugar na hierarquia de classes.
- Interfaces são úteis para:
  - Capturar similaridades entre classes não relacionadas.
  - Declarar métodos que uma ou mais classes devem inevitavelmente implementar.
  - Revelar interfaces sem revelar os objetos que a implementam (útil na venda de pacotes de componentes).

# Métodos default

- Até o Java 7, interface não podia prover nenhuma implementação.
- No Java 8, um método *default* permite definir um método de interface com implementação.
- Permite expandir a interface sem violar o código existente.
- Permite implementar métodos que são opcionais, dependendo da forma como a interface é usada.
- Pode produzir erro de herança múltipla de método.

# Métodos default

```
public interface IFace {  
    // decl. método normal  
    int getUserID();  
    // decl. método default  
    default int getAdminID() {  
        return 1;  
    }  
}  
  
class Demo {  
    public static void main(String args[]) {  
        IFaceImpl obj = new IFaceImpl();  
  
        System.out.println("User ID is " + obj.getUserID());  
        System.out.println("Administrator ID is " + obj.getAdminID());  
    }  
}
```

<sup>1</sup>Adaptado de: Herbert Schildt. *Java Para Iniciantes*. Bookman 2015.



# Métodos default

```
public interface Ordenavel {  
  
    boolean menorQue(Ordenavel o);  
  
    boolean igual(Ordenavel o);  
  
    default boolean diferente(Ordenavel o) {  
        return !igual(o);  
    }  
  
    default boolean maiorQue(Ordenavel o) {  
        return !menorQue(o) && !igual(o);  
    }  
  
}
```

# Expressões Lambda e interfaces funcionais

- Expressões lambda e interfaces funcionais são elementos da programação funcional incorporados ao Java.
- Programação funcional, com sua ênfase em funções "puras", tratadas como valores de 1ª classe (que retornam o mesmo resultado dadas as mesmas entradas, sem a produção de efeitos colaterais) e a imutabilidade simplificam a programação paralela.

## Interfaces funcionais (*functional interface*)

são interfaces com um único método abstrato. Sua implementação pode ser feita por uma classe regular, classe interna, classe anônima ou expressão lambda.

# Expressões Lambda e interfaces funcionais

- 1 Considere a interface `Comparator<T>`, disponível no Java.
- 2 Ela é uma *functional interface* baseada no método `compare`, e pode ser implementada por uma expressão lambda ou referência de método.
- 3 Ela possui inúmeros métodos default e outros métodos que já são implementados a partir da classe `Object`.

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);

    ...
}
```

# Interfaces funcionais com classes internas

- 1 Considere a necessidade de se ordenar o estoque de produtos pela quantidade disponível.
- 2 Solução possível: classe `OrdenarPorQuantidade` implementa a interface `Comparator<T>`.

```
private class OrdenarPorQuantidade implements Comparator<Produto> {  
  
    @Override  
    public int compare(Produto o1, Produto o2) {  
        return Integer.compare(o1.getQuantidade(), o2.getQuantidade());  
    }  
}  
  
public void ordenarPorQuantidade() {  
    Arrays.sort(produtos, 0, numProdutos, new OrdenarPorQuantidade());  
}
```

# Interfaces funcionais com classes anônimas

- 1 Considere a necessidade de se ordenar o estoque de produtos pela Data de Fabricação.
- 2 Solução possível: classe interna anônima que implementa a interface `Comparator<T>`.

```
public void ordenarPorFabricacao() {  
    Arrays.sort(produtos, 0, numProdutos, new Comparator<Produto>() {  
        @Override  
        public int compare(Produto o1, Produto o2) {  
            return o1.getDataFabricacao().compareTo(o2.getDataFabricacao());  
        }  
    });  
}
```

# Interfaces funcionais com expressões lambda

- 1 Considere a necessidade de se ordenar o estoque de produtos pelo preço.
- 2 Solução possível: expressão lambda que implementa a interface `Comparator<T>`.

```
public void ordenarPorPreco() {  
    Arrays.sort(produtos, 0, numProdutos, (o1, o2) -> Float.compare(o1.  
})
```