Programação Orientada a Objetos

Um objeto é a representação computacional de uma entidade do mundo real, com:

• atributos (necessariamente) privados

• operações

• Objeto = Dados Privados (variáveis de instância) + Operações (métodos)

Definição de Objeto:

• independência do contexto (reutilização)

• abstração de dados (abstração)

• encapsulamento (abstração e privacidade)

• modularidade (composição)

Encapsulamento

• Um objeto deve ser visto como uma “cápsula”, assegurando a proteção dos dados internos

Dados Privados

(v. instância)

método 1

método 2

método 3

método 4

método privado

Um objeto é:

• uma unidade computacional fechada e autónoma

• capaz de realizar operações sobre os seus atributos internos

• capaz de devolver respostas para o exterior, sempre que estas lhe sejam solicitadas

• capaz de garantir uma gestão autónoma do seu espaço de dados interno

Regras de acesso a variáveis e métodos

• a declaração deve ser complementada com informação sobre o nível de visibilidade das variáveis e métodos.

- Public: A partir de qualquer classe
- Private: Apenas acessível dentro da classe
- Protected: Acessível a partir da classe, de classes do mesmo package e de todas as subclasses
- Default: acessível a partir da classe e classes do mesmo package

• para garantir o total encapsulamento do objeto as variáveis de instância devem ser declaradas como private

• ao ter encapsulamento total é necessário garantir que existem métodos que permitem o acesso e modificação das variáveis de instância.

• os métodos que se pretendem que sejam visíveis do exterior devem ser declarados como public

Breve introdução ao Diagrama Classes UML

- Visibilidade de atributos e operações

- Privado

+ Publico

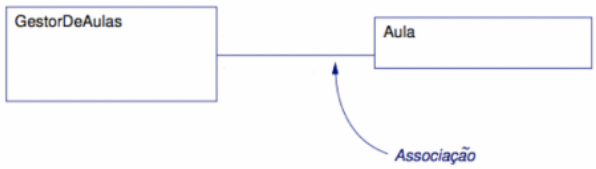
Package (Sem nada)

Protegido

- Relação entre classes – dependência

- Indica que a definição de uma classe está dependente da definição de outra.
- Utiliza-se normalmente para mostrar que instâncias da origem utilizam, de alguma forma, instâncias do destino (por exemplo: um parâmetro de um método)
- Uma alteração no destino (quem é usado) pode alterar a origem (quem usa)

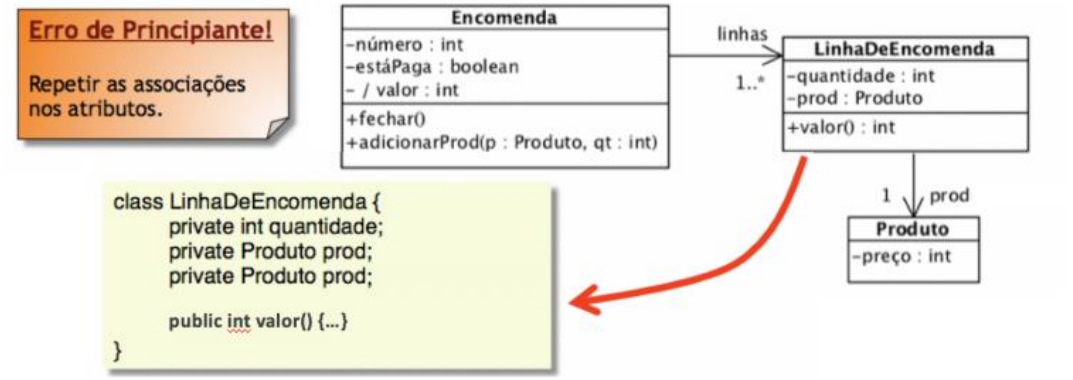
- Relações entre classes - Associação

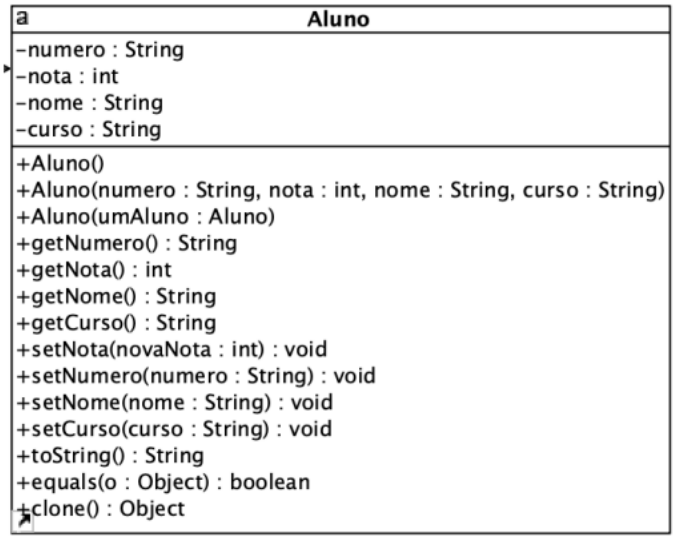


- Indica que objectos de uma estão ligados a objectos de outra — define uma relação entre os objectos
- Noção de navegabilidade (cf. diagramas E-R)
- Por omissão representam navegação bidireccional — mas pode indicar-se explicitamente o sentido da navegabilidade.

- Associações vs Atributos

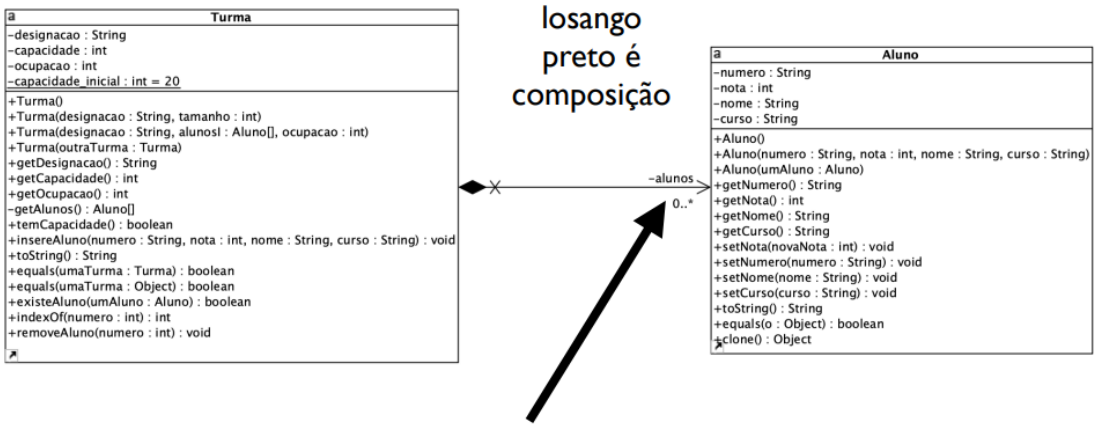
- Atributos (de instância) representam propriedades das instâncias das classes
  - São codificados como variáveis de instância
- Associações também representam propriedades das instâncias das classes
  - também são codificados como variáveis de instância
- Atributos devem ter tipos simples
  - **utilizar associações para tipos estruturados**





variáveis de instância

construtores + métodos de instância



a v.i. chama-se alunos, é privada e pode ter zero ou mais instâncias de Aluno

Clone vs Encapsulamento

• a utilização de clone() permite que seja possível preservarmos o encapsulamento dos objetos, desde que:

• seja feita uma cópia dos objetos à entrada dos métodos

• seja devolvida uma cópia dos objetos e não o apontador para os mesmos

Igualdade de objectos

• alunos[i] == a, não é eficaz porque compara os apontadores (e pode ter havido previamente um clone)

• (alunos[i]).getNumero() == a.getNumero(), assume demasiado sobre a forma como se comparam alunos

O método equals

A relação de equivalência que o método implementa é:

• é reflexiva, ou seja x.equals(x) == true, para qualquer valor de x que não seja nulo

• é simétrica, para valores não nulos de x e y se x.equals(y) == true, então y.equals(x) == true

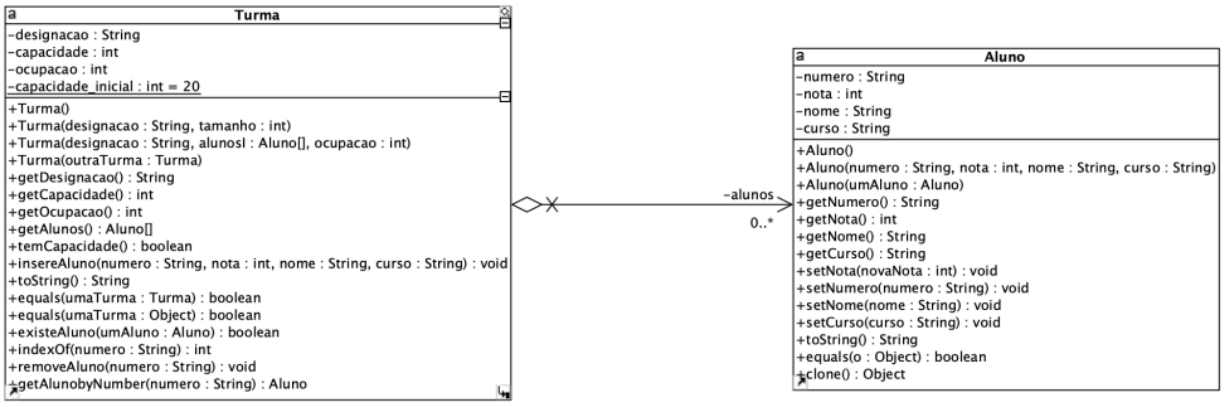
• é transitiva, em que para x,y e z, não nulos, se x.equals(y) == true, y.equals(z)

== true, então x.equals(z) == true

• é consistente, dado que para x e y não nulos, sucessivas invocações do método equals (x.equals(y) ou y.equals(x)) dá sempre o mesmo resultado

• para valores nulos, a comparação com x,não nulo, dá como resultado false.

A arquitetura com agregação:



… em resumo

• Se o diagrama de classes indicar uma associação de composição:

• faz-se uma cópia (clone) dos objetos quando são guardados internamente

• devolve-se sempre uma cópia dos objetos e, caso seja necessário, da estrutura de dados que os guarda

… em resumo

• Se o diagrama de classes indicar uma associação de agregação:

• guarda-se internamente o apontador dos objetos passados como parâmetro

• devolve-se sempre o apontador dos objetos e, caso seja solicitado, uma cópia da estrutura de dados que os guarda

… em resumo

• Quando o diagrama de classes não explicitar se a associação é de composição ou de agregação, parte-se do princípio que é de composição!

• O mesmo se aplica quando não se fornece o diagrama de classes.

Composição ou agregação?

• Ao criar esta estrutura temos de tomar a decisão se a coleção de elementos assenta em composição ou agregação.

  • Quais são as implicações?

• Se for composição os métodos de inserção e get terão de prever a utilização de clone

  • é a solução adequada para todas as situações?

• Se a estratégia de construção for do tipo Composição, esta coleção terá sempre este comportamento.

• não poderá ser utilizada em estratégias de agregação.

• os métodos de get e set, adição e recuperação (e os construtores) farão sempre clone e gerarão novas referências.

Coleções Java

• O Java oferece um conjunto de classes que implementam as estruturas de dados mais utilizadas

• oferecem uma API consistente entre si

• permitem que sejam utilizadas com qualquer tipo de objecto – são parametrizadas por tipo

• Poderemos representar:

• ArrayList<Aluno> alunos

• HashSet<Aluno> alunos;

• HashMap<String, Aluno> turmaAlunos;

• TreeMap<String, Docente> docentes;

• Stack<Pedido> pedidosTransferência;

• ...

• Ao fazer-se ArrayList<Aluno> passa a ser o compilador a testar, e validar, que só são utilizados objetos do tipo Aluno no ArrayList.

• isto dá uma segurança adicional aos programas, pois em tempo de execução

não teremos erros de compatibilidade de tipos

• os tipos de dados são verificados em tempo de compilação

• As coleções em Java beneficiam de:

• auto-boxing e auto-unboxing, ie, a capacidade de converter automaticamente tipos primitivos para instâncias de classes wrapper.

• int para Integer, double para Double,

etc.

• o programador não tem de codificar a transformação

• tipos genéricos

• as coleções passam a ser definidas em função de um tipo de dados que é

associado aquando da criação

• a partir daí o compilador passa a garantir que os conteúdos da coleção são do tipo esperado

Iteradores externos

• O Iterator é um padrão de conceção bem conhecido e que permite providenciar uma forma de aceder aos elementos de uma coleção de objetos, sem que seja necessário saber qual a sua representação interna

• basta para tal, que todas as coleções saibam criar um iterator!

• não precisamos saber como tal é feito!

• Um iterador de uma lista poderia ser:

• o iterator precisa de ter mecanismos para:

• aceder ao objeto apontado

• avançar

• determinar se chegou ao fim

• Utilizando Iterators...

```
/**
 * Algum aluno passa?
 *
 * @return true se algum aluno passa
 */
public boolean alguemPassa() {
    boolean alguem = false;
    Iterator<Aluno> it = lstAlunos.iterator();
    Aluno a;

    while(it.hasNext() && !alguem) {
        a = it.next();
        alguem = a.passa();
    }
    return alguem;
}
```

```
/**
 * Remover notas mais baixas
 *
 * @param nota a nota limite
 */
public void removerPorNota(int nota) {
    Iterator<Aluno> it = lstAlunos.iterator();
    Aluno a;

    while(it.hasNext()) {
        a = it.next();
        if (a.getNota() < nota)
            it.remove();
    }
}
```

Iterator<E>

• Em resumo...

• Todas as coleções implementam o método: Iterator<E> iterator() que cria um iterador ativo sobre a coleção

• Padrão de utilização:

```
Iterator<E> it = colecção.iterator();
E elem;

while(it.hasNext()) {
    elem = it.next();
    // fazer algo com elem
}
```

• Procurar:

```
boolean encontrado = false;
Iterator<E> it = colecção.iterator();
E elem;

while(it.hasNext() && !encontrado) {
    elem = it.next();
    if (criterio de procura sobre elem)
        encontrado = true;
}
// fazer alguma coisa com elem ou com encontrado
```

• Remover:

```
Iterator<E> it = colecção.iterator();
E elem;

while(it.hasNext()) {
    elem = it.next();
    if (criterio sobre elem)
        it.remove();
}
```

• Iterador externo

```
/**
 * Subir a nota a todos os alunos
 *
 * @param bonus int valor a subir.
 */
public void aguaBenta(int bonus) {
    for(Aluno a: lstAlunos)
        a.sobeNota(bonus);
}
```

• Iterador interno forEach()

```
/**
 * Subir a nota a todos os alunos
 *
 * @param bonus int valor a subir.
 */
public void aguaBenta(int bonus) {
    lstAlunos.forEach((Aluno a) -> {a.sobeNota(bonus);});
}
```

Expressões Lambda

```
/**
 * Subir a nota a todos os alunos
 *
 * @param bonus int valor a subir.
 */
public void aguaBenta(int bonus) {
    lstAlunos.forEach(a -> a.sobeNota(bonus));
}
```

Streams

```
public long quantosPassam() {

    return lstAlunos.stream().filter(a -> a.passa()).count();
}
```

Criação de estruturas ordenadas

• Criar um TreeSet de Aluno com ordenação por comparador

```
TreeSet<Aluno> alunos = new TreeSet<>(new ComparatorAlunoNome());
```

• Criar um TreeSet com a comparação dada pela ordem natural:

```
TreeSet<Aluno> turma = new TreeSet<>();
```

• Criar um TreeSet definido o comparator do mesmo na invocação (via classe anónima). Excessivamente complicado!

```
TreeSet<Aluno> teóricas = new TreeSet<>(
                          new Comparator<Aluno>() {
                             public int compare(Aluno a1, Aluno a2) {
                               return a1.getNome().compareTo(a2.getNome());
                             }
                          });
```

• Uma outra forma é recorrer a um método anónimo, escrito sob a forma de uma expressão lambda.

```
TreeSet<Aluno> praticas = new TreeSet<>((a1,a2) ->
                                        a1.getNome().compareTo(a2.getNome()));
```

• ou, se quisermos reutilizar as expressões:

```
Comparator<Aluno> comparador = (a1, a2) -> a1.getNome().compareTo(a2.getNome());

TreeSet<Aluno> tutorias = new TreeSet<>(comparador);
```

Herança vs Composição

• Herança e composição são duas formas de relacionamento entre classes

• são no entanto abordagens muito distintas e constitui um erro muito comum achar que podem ser utilizadas da mesma forma

• existe uma tendência para se confundir herança com composição

• quando uma classe é criada por composição/agregação de outras, isso implica que as instâncias das classes compostas/agregadas fazem parte da definição do contentor

• é uma relação do tipo “parte de” (partof)

• qualquer instância da classe vai ser constituída por instâncias das classes compostas/agregadas

• Exemplo: Círculo tem um ponto central

• do ponto de vista do ciclo de vida a relação é fácil de estabelecer:

• (quando é composição) quando a instância contentor desaparece, as instâncias agregadas também desaparecem

• o seu tempo de vida está iminentemente ligado ao tempo de vida da instância de que fazem parte!

• (quando é agregação) desaparece a relação entre os objetos

• esta é uma forma (…e está aqui a confusão!) de criar entidades mais complexas a partir de entidades mais simples:

• Turma é composta por instâncias de Aluno

• Automóvel é composto por Pneu, Motor, Chassis, …

• Clube é composto por instâncias de Atleta, Funcionário, Dirigente, ...

• quando uma classe (apesar de poder ter instâncias de outras classes no seu estado interno) for uma especialização de outra, então a relação é de herança

• quando não ocorrer esta noção de especialização, então a relação deverá ser

de composição/agregação

• Uma forma simples de testar se faz sentido a relação ser de herança é “ler” o diagrama.

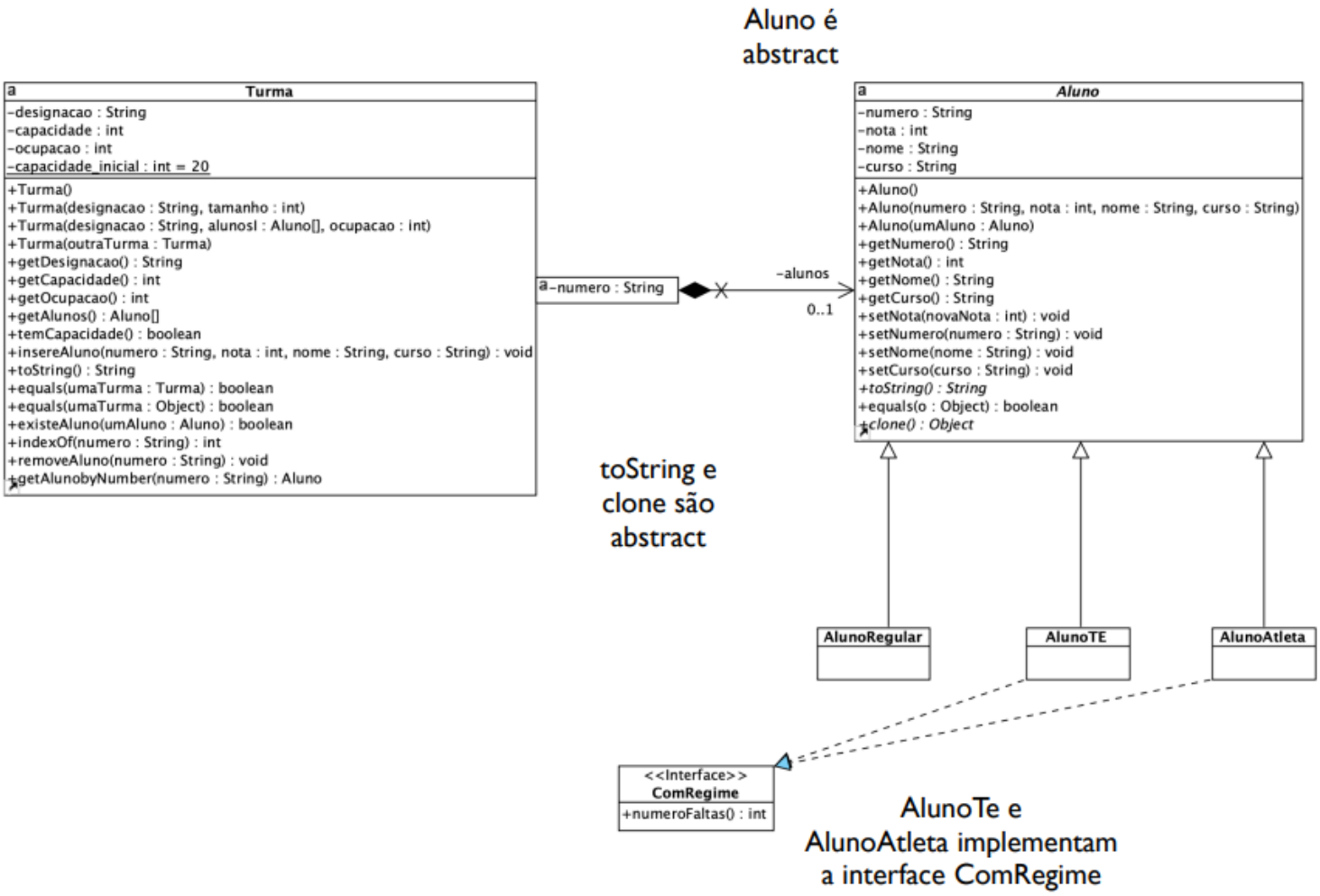
Em resumo...

• As interfaces Java são especificações de tipos de dados. Especificam o conjunto de operações a que respondem objetos desse tipo

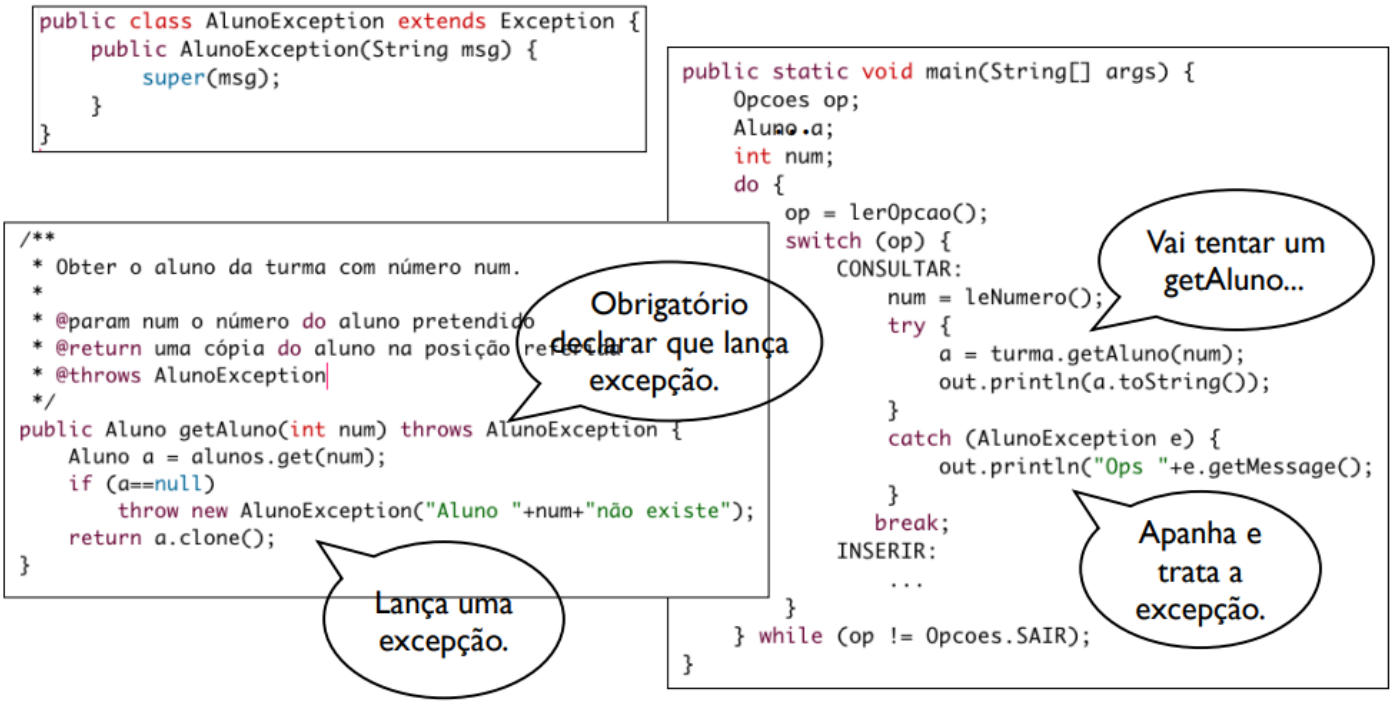
• Uma instância de uma classe é imediatamente compatível com:

  • o tipo da classe

  • o tipo da interface (se estiver definido)



Criar Excepções



Leitura/Escrita em ficheiros

• Gravar em modo texto:

```
/**
 * Método que guarda o estado de uma instância num ficheiro de texto.
 *
 * @param nome do ficheiro
 */

public void escreveEmFicheiroTxt(String nomeFicheiro) throws IOException {
   PrintWriter fich = new PrintWriter(nomeFicheiro);
   fich.println("------- HotéisInc --------");
   fich.println(this.toString()); // ou fich.println(this);
   fich.flush();
   fich.close();
 }
```

• Leitura em modo binário

• utilização de java.io.ObjectInputStream

```
/**
 * Método que recupera uma instância de HoteisInc de um ficheiro de objectos.
 * Este método tem de ser um método de classe que devolva uma instância já
 * construída de HoteisInc.
 *
 * @param nome do ficheiro onde está guardado um objecto do tipo HoteisInc
 * @return objecto HoteisInc inicializado
 */

public static HoteisInc carregaEstado(String nomeFicheiro) throws FileNotFoundException,
                                        IOException,
                                        ClassNotFoundException {
    FileInputStream fis = new FileInputStream(nomeFicheiro);
    ObjectInputStream ois = new ObjectInputStream(fis);
    HoteisInc h = (HoteisInc) ois.readObject();
    ois.close();
    return h;
}
```