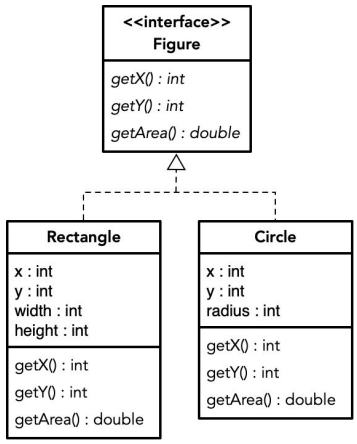
Módulo 6 - Herança e classes abstratas

Objetivos

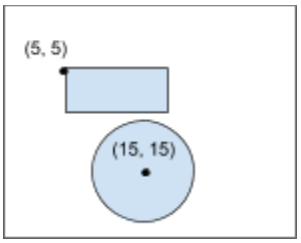
- Entender as vantagens do mecanismo de herança.
- Saber desenvolver classes abstratas e derivadas.
- Entender a importância de manter o contrato em classe derivadas.

Motivação: polimorfismo sem duplicação de código

No contexto de uma aplicação que manipula figuras geométricas, suponha que existe a seguinte interface, a implementar nas várias classes que representam os diferentes tipos figuras, que define as operações para a localização e cálculo da área.




```
interface Figure {
  int getX();
  int getY();
  double getArea();
}
```



Em seguida são apresentadas duas classes para implementar as figuras retângulo e círculo, obedecendo à interface acima. Ambas as classes têm os mesmos atributos para a localização (x, y), e os métodos inspetores são também iguais.

```
class Rectangle implements Figure {
  private int x;
  private int y;
  private int width;
  private int height;
  ...
  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

  public double getArea() {
    return width * height;
  }
}
```

```
class Circle implements Figure {
  private int x;
  private int y;
  private int radius;
  ...
  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

  public double getArea() {
    return Math.PI * radius * radius;
  }
}
```

O poliformismo do tipo Figure é alcançado, porém, as implementações têm elementos iguais, o que deu origem a algum código duplicado. Outras classes para outros tipos de figura teriam também que duplicar a parte do código relativa à localização. O polimorfismo oferece benefícios relativos ao desenvolvimento de código mais genérico, porém, a duplicação de código é um fenómeno que deve ser evitado. O mecanismo de herança facilita esta questão, permitindo que uma classe seja baseada noutra, reutilizando a sua implementação.

Classes abstratas

Uma das formas de aplicar herança é baseada na noção de *classe abstrata*. Uma classe abstrata pode ser vista como uma classe incompleta, onde faltam definições de métodos, por oposição a uma *classe concreta*. Tais métodos em falta podem ser derivados no facto da classe abstrata implementar uma interface de forma incompleta (i.e., não definir todas as operações da mesma). Outra possibilidade é a classe abstrata declarar operações (métodos) sem definir em concreto a sua implementação, de forma análoga às interfaces (ilustrada mais à frente).

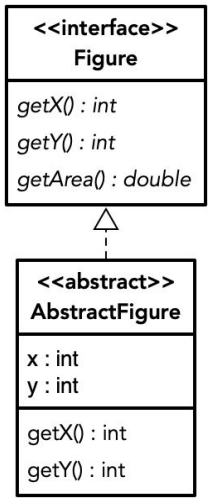
Para abordar o problema da duplicação apresentado na secção anterior, podemos definir uma classe abstrata AbstractFigure que implementa as partes comuns das classes Rectangle e Circle. Uma classe abstrata é definida utilizando o modificador *abstract*. Embora a classe AbstractFigure declare que implementa a interface Figure e não implemente uma das suas operações (getArea()), a mesma é válida, dado que por ser abstrata pode ter operações em falta.

```
public abstract class AbstractFigure implements Figure {
  private int x;
  private int y;

  public AbstractFigure(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }
}
```



Uma classe abstrata não é instanciável, pois está incompleta. Mesmo que a classe não tenha operações em falta e que defina um construtor, o facto de ter sido declarada abstrata impossibilitará a criação de objetos.

```
AbstractFigure f = new AbstractFigure(50, 100);
```

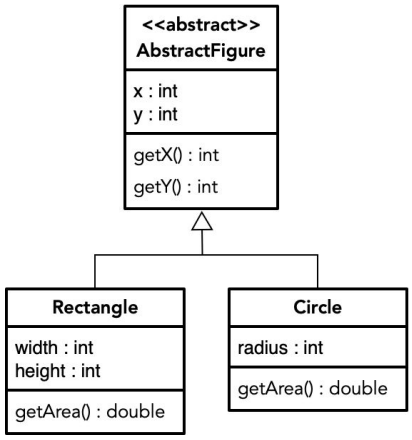
Classes derivadas

O propósito de ter uma classe abstrata é esta ser a *base* (ou *superclasse*) para a definição de *classes derivadas* (ou *subclasses*), as quais definirão as operações em falta. Ao evoluir o exemplo anterior, as classes Rectangle e Circle serão definidas como *derivadas* da classe AbstractFigure, por via da diretiva *extends*, herdando todos os seus atributos e operações. Caso a classe derivada não seja ela também abstrata, terá que definir os métodos em falta na classe base (getArea() neste exemplo). Uma classe só pode derivar de uma única classe base, podendo esta ser abstrata ou não. As relações de derivação são transitivas, significando que se *A* deriva de *B*, e *B* deriva de *C*, *A* herda os membros de *B* e de *C*.

```
public class Rectangle extends AbstractFigure {
  private int width;
  private int height;

  public Rectangle(int x, int y, int width, int height) {
    super(x, y);
    this.width = width;
    this.height = height;
  }

  public double getArea() {
    return width * height;
  }
}
```



Os construtores não são herdados da classe base. A definição de uma classe derivada obriga a que em todos os seus construtores seja feita uma invocação a um dos construtores da classe base como primeira instrução, através da chamada *super(...).* Caso a classe base tenha um construtor sem argumentos, a instrução de invocação pode ser omitida (porém executará implicitamente). No exemplo acima o construtor da classe base (AbstractFigure) é invocado com os argumentos (x, y). A obrigatoriedade da invocação do construtor da classe base serve para garantir que os atributos definidos na mesma sejam inicializados com valores válidos (que é o papel dos construtores).

À semelhança das interfaces, a relação entre uma classe derivada e uma classe base possibilita a utilização de referências polimórficas do tipo base, as quais são compatíveis com todas as classes derivadas (direta ou indiretamente).

```
AbstractFigure f = new Rectangle(...);
double area = f.getArea();
```

Exercício 6.1 - Propriedades de figuras geométricas

a) Utilize a interface Figure apresentada anteriormente (bem como AbstractFigure), declarando uma operação adicional para obter o perímetro da figura.
b) Complete a classe Rectangle, tendo em conta a nova operação.
c) Desenvolva a classe Circle para representar círculos.
d) Desenvolva uma classe Canvas, que contém um conjunto de figuras. Deverá ser possível:
   i) Adicionar uma figura;
   ii) Remover uma figura;
   iii) Remover a figura que têm a área maior (se existir pelo menos uma figura);
   iv) Saber o total das áreas das figuras contidas.

Métodos abstratos

Uma classe abstrata pode definir métodos abstratos, os quais são equivalentes às declarações de operações nas interfaces. Os métodos abstratos terão que ser definidos pelas classes derivadas (que sejam concretas). O seguinte exemplo apresenta uma implementação do algoritmo de ordenação *bubble sort* abstrato, pois o método que decide se dois números estão por ordem é abstrato (não está implementado). Desta forma, as várias políticas de ordenação poderão ser implementadas em classes derivadas que definem o método abstrato.

```
public abstract class BubbleSorter {
  public final void sort(int[] v) {
    for(int i = 0; i < v.length; i++)
      for(int j=1; j < v.length-i; j++)
      if(isBefore(v[j], v[j-1])) {
        int temp = v[j-1];
            v[j-1] = v[j];
            v[j] = temp;
      }
  }

  public abstract boolean isBefore(int a, int b);
}
```

```
public class BubbleSorterAsc
extends BubbleSorter {
  public boolean isBefore(int a, int b) {
    return a < b;
  }
}
```

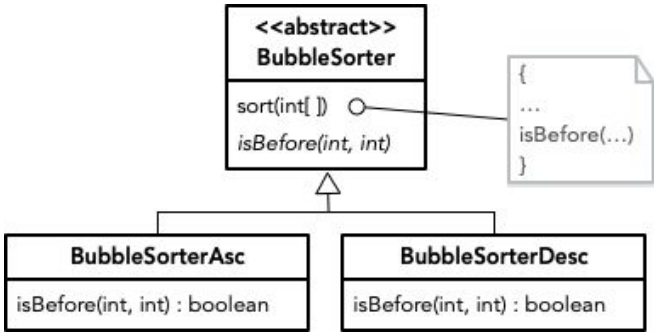
```
public class BubbleSorterDesc
extends BubbleSorter {
  public boolean isBefore(int a, int b) {
    return a > b;
  }
}
```

O algoritmo poderia ser utilizado instanciando uma classe derivada de BubbleSorter:

```
int[] v = {4, 1, 2, 3, 0};
BubbleSorter sorter = new BubbleSorterAsc();
sorter.sort(v);         // {0, 1, 2, 3, 4}
```

Padrão Método Incompleto (*Template Method*)

A solução apresentada acima é uma aplicação do padrão de desenho Método Incompleto (*Template Method*). Este tipo de solução consiste em ter uma classe abstrata que contém métodos que invocam métodos abstratos da própria classe, e daí incompletos, porque se referem a algo que não está disponível. Este padrão é uma alternativa ao padrão Estratégia abordado anteriormente. Porém, a solução com uma estratégia é mais flexível, dado que o comportamento do objeto que executa o algoritmo pode variar em tempo execução (a componente variável é passada como parâmetro).



Exercício 6.2 - Leitor abstrato de objetos de ficheiro

Desenvolva uma classe abstrata para representar um leitor de ficheiros de texto que instancia um objeto de um tipo T (em aberto) por cada linha de texto do ficheiro. A criação dos objetos a partir da String (linha) deverá ficar representada num método abstrato. Para testar a classe abstrata, desenvolva duas classes derivadas para ler ficheiros com linhas de objetos Aluno (número e nome) e Time (exercício 4.2).

```
public abstract class LineObjectReader<T> {
  ...
  public LineObjectReader(File file) {...

  public List<T> read() throws FileNotFoundException {
    ...
```

Sobreposição de métodos

Nos exemplos anteriores, as classes derivadas estão a completar a classe base (abstrata), definindo os métodos em falta. Porém, uma classe derivada pode também *sobrepor* um método definido na classe base (que chamaremos de método base). Uma *sobreposição* de método funciona como uma substituição do método da classe base, sendo fornecida outra implementação para o mesmo. Desta forma, ao ser invocada uma operação num objeto de uma classe derivada, será executado o método base, caso este não tenha sido sobreposto, ou o método sobreposto na classe derivada. Perante casos em que um método tenha sido sobreposto, mas onde seja necessário utilizar a implementação base, as classes derivadas podem invocar o método base fazendo *super.metodo(...)*.

A classe java.util.ArrayList permite a inserção de referências nulas, o que pode ser considerada uma prática indesejada. O seguinte exemplo apresenta uma classe derivada de ArrayList que sobrepõe o método add, por forma a não permitir a inserção de referências nulas. Caso o argumento seja uma referência seja nula é lançada uma exceção, caso contrário a execução prossegue com a implementação base.

```
public class ArrayListNoNulls<E> extends ArrayList<E> {
  @Override
  public boolean add(E element) {
    if(element == null)
      throw new IllegalArgumentException("cannot add null");
    return super.add(element);
  }
  ...
}
```

A anotação @Override pode ser utilizada *opcionalmente* num método, por forma a que seja verificado que o método corresponde realmente a uma sobreposição (caso contrário é gerado um erro de compilação). A inclusão desta anotação nunca terá impacto na execução, serve apenas para confirmar a intenção do programador, e por via do erro alertar para o engano. Uma classe pode impedir que um método seu seja sobreposto utilizando o modificador *final* na sua declaração (como no método sort(...) apresentado anteriormente). Os construtores não se podem sobrepor (nem tão pouco são herdados, como mencionado anteriormente).

Exercício 6.3 - Empregados

Crie um programa que use uma lista de empregados para calcular o total dos salários a pagar de uma cadeia de lojas. A lista de empregados deve conter empregados (sem especialização), gerentes de loja e diretores regionais. Para cada classe de empregados, o salário é calculado da seguinte maneira:

- Empregados: valor fixo de 800€;
- Gerente de loja: valor fixo igual ao dos empregados sem especialização, acrescido de um prémio de 200€ que é atribuído sempre que a loja cumpre os objectivos das vendas;
- Diretor regional: valor fixo igual ao dobro do dos empregados sem especialização, acrescido de um prémio que corresponde a 1% do lucro mensal nas lojas da região.

Nota: para simplificar, considere que ter cumprido ou não os objectivos de vendas é um atributo dos gerentes e que o lucro mensal da região é um atributo dos diretores.

Subtipos

A herança é um mecanismo de reutilização de código, porém devemos ter atenção às relações classe base-derivada para não incorrer em problemas. Uma classe D derivada de uma classe base B deverá corresponder a uma relação de subtipo (“é um”), de forma a que faça sentido afirmar “D é um B”. Como contra-exemplo, suponha uma classe para representar elipses como sendo derivada de uma classe para representar círculos, no sentido em que “uma elipse é um círculo com uma dimensão extra para além do diâmetro”. Considere a seguinte implementação do exemplo descrito. Qual o sentido da operação getDiameter() para uma elipse, o que deve devolver? A operação não faz sentido, porque o conceito de diâmetro não se aplica a uma elipse, porque uma elipse não é um círculo.

```
public class Circle {
  private int diameter;

  public Circle(int diameter) {
    this.diameter = diameter;
  }

  public int getDiameter() {
    return diameter;
  }
}
```

```
public class Ellipse extends Circle {
  private int height;

  public Ellipse(int width, int height) {
     super(width);
     this.height = height;
  }
}
```

Ao violar a relação de subtipo na aplicação de herança, facilmente se obtêm classes cujas operações (herdadas) não fazem sentido. Desta forma, em termos de subtipos fará sentido afirmar que um círculo é uma elipse cuja altura e largura são iguais. O seguinte exemplo ilustra

esta solução com uma classe de objetos mutáveis para representar elipses, e uma classe derivada para círculos.

```
public class Ellipse {
  private int width;
  private int height;

  public Ellipse(int width, int height) {
     this.width = width;
     this.height = height;
  }
  ...
}
```

```
public class Circle extends Ellipse {
  public Circle(int diameter) {
    super(diameter, diameter);
  }

  public int getDiameter() {
    return getWidth();
  }
}
```

Porém, esta solução embora respeite a relação de subtipo, não está livre de problemas. O facto da classe base ser de objetos mutáveis, a alteração de uma das dimensões fará com que o círculo fique com um estado incoerente face à sua propriedade da altura ser igual à largura.

```
Circle c = new Circle(10);    // c.getWidth() == c.getHeight()
c.setWidth(20);               // c.getWidth() != c.getHeight()
```

Caso os objetos fossem imutáveis este problema não se colocaria, pois após a criação do objeto círculo as dimensões não seriam alteradas. Com objetos mutáveis, o problema poderia ser contornado sobrepondo os métodos setWidth(...) e setHeight(...) na classe Circle por forma a que ambas as dimensões sejam alteradas com o mesmo valor (mantendo a propriedade altura igual a largura).

Quando os atributos da classe base têm mais informação do que a necessária (ou redundante) para representar os objetos da classe derivada, para além da questão da utilização desnecessária de memória, estamos perante um indício que o desenho das classes terá que ser feito cuidadosamente para garantir a coerência (como no exemplo dado). Por outro lado, quando uma classe derivada não acrescenta atributos face à classe base, provavelmente não se justifica a existência da classe derivada, pois o que a mesma pretende representar pode ser definido através de uma propriedade (função) na classe base. No exemplo dado, a classe Ellipse poderia incluir um método isCircle() para saber se a elipse corresponde a um círculo.

Fragilidade da herança

Ao não utilizar nenhum modificador de acesso nos membros da classe (*package-private*), estes serão acessíveis dentro do mesmo pacote. Isto significa que uma classe derivada cuja classe base se encontra no mesmo pacote pode aceder a estes membros. O Java oferece uma outra possibilidade de acesso um pouco mais permissiva através do modificador *protected*, que

permite o acesso dentro do mesmo pacote e adicionalmente às classes derivadas (que poderão estar noutro pacote).

Uma classe que é desenhada para ser extensível por herança facilmente consiste numa classe frágil, pois não é fácil desenhar a mesma de forma a que as classes derivadas não comprometam a coerência da funcionalidade definida na classe base. Se uma classe base necessita de expor a sua lógica interna às classes derivadas, então a aplicação de herança quebra o encapsulamento, o que não é desejável. Dada a fragilidade da herança, as hierarquias de classes são normalmente definidas dentro de um mesmo pacote, fazendo com a eventual quebra de encapsulamento fique confinada ao pacote em questão.

Quando uma classe não foi desenhada para ser estendida é possível (e recomendado) não permitir classes derivadas utilizando o modificador *final* na declaração da classe.

```
public final class NonExtensibleClass { ...
```

Quando uma classe é desenhada para ser extensível, os atributos deverão ser privados, seguindo a ideia de encapsulamento. Embora por vezes possa ser conveniente numa classe derivada aceder a atributos da classe base, não deverá ser permitido modificar os mesmos na classe derivada, pois poderá ser comprometida a coerência dos atributos da classe base.

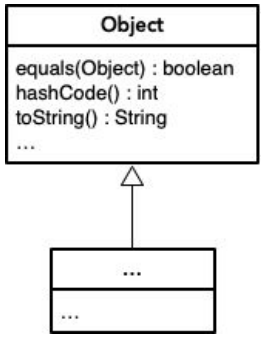
Classe java.lang.Object

As classes em Java são todas forçosamente derivadas de uma classe base comum - a classe java.lang.Object. Desta forma, e ainda que não seja necessário declará-lo, todas as classes estão na prática definidas como sendo classes derivadas de Object, a não ser que se declarem como sendo derivadas de outra classe.

```
public AnyClass extends Object { ...
```

Assim sendo, uma referência do tipo Object é polimórfica e compatível com todos os objetos de quaisquer classes.

```
Object o = ...  // qualquer objeto
```



Esta classe define um conjunto de métodos, herdados por todas as classes, que têm grande importância na biblioteca das coleções, assim como em outras classes de utilitários e classes que intervêm na programação concorrente. Nesta secção analisaremos apenas os seguintes:

- equals(Object) : boolean, devolve verdadeiro caso o objeto (instância) seja considerado igual a outro apontado pela referência do argumento. A implementação base devolve verdadeiro caso o objeto seja o *mesmo* (ao contrário de uma comparação de conteúdo). Por outras palavras, se a referência aponta para o objeto onde é invocado o método.

- hashCode() : int, devolve um inteiro que representa um *hash* ("resumo") do objeto. A implementação base devolve um inteiro diferente para cada objeto em memória, sendo esse constante durante a execução do programa e relacionado com o endereço de memória onde está guardado o objeto.

- toString() : String, devolve uma string que descreve o objeto textualmente. A implementação base devolve o nome da classe do objeto seguido do resultado de hashCode() em hexadecimal. A implementação base não é muito útil, e é aconselhado que as classes a sobreponham.

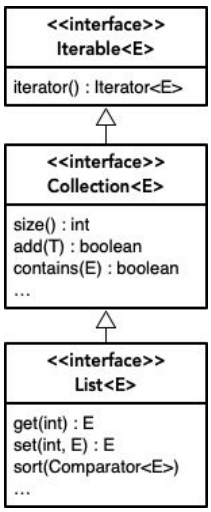
Os métodos equals(...) e hashCode() estão disponíveis para todos os objetos, pois foram desenhados com o propósito de serem utilizados nas classes da biblioteca de coleções. Por exemplo, a implementação das operações de contains(...) na classes para listas (p.e. ArrayList) baseiam-se no resultado de equals(...) para determinar se o objeto da procura está contido na lista. Estes métodos podem ser sobrepostos, mas isso deverá ser feito com precaução.

Herança em interfaces

Uma interface pode ser derivada de uma ou mais interfaces, significando que a interface *herda* operações definidas por essa(s) interface(s) base. Este mecanismo é útil quando estamos perante um conjunto de interfaces relacionadas que têm operações em comum. O exemplo seguinte apresenta um excerto de interfaces das bibliotecas do Java, java.util.Iterable, java.util.Collection, e java.util.List, as quais derivam sucessivamente umas das outras.

```
public interface Collection<T> extends Iterable<T> {
  int size();
  boolean add(T element);
  boolean contains(T element);
  ...
}

public interface List<T> extends Collection<T> {
  T get(int index);
  T set(int index, T element);
  void sort(Comparator<T> comparator);
  ...
}
```



Uma interface pode definir implementações por omissão para as suas operações, utilizando o modificador *default*. Estas implementações são herdadas nas classes que implementam as interfaces, as quais não são obrigadas a fornecer implementações para estas operações, mas podem podem porém sobrepor as implementações base. A possibilidade de fornecer implementações por omissão é particularmente útil quando a implementação de uma operação se consegue concretizar apenas com base noutras operações declaradas na interface, sem aceder diretamente a atributos (pois estes não podem ser definidos nas interfaces). Este exemplo é uma aplicação do padrão método incompleto explicado anteriormente.

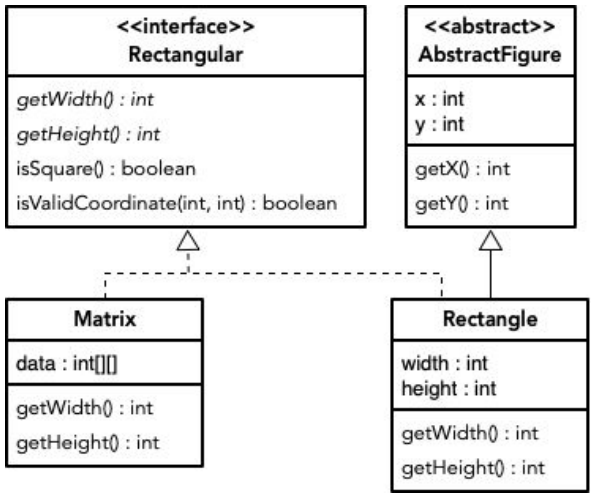
```
public interface Rectangular {
  int getWidth();
  int getHeight();

  default boolean isSquare() {
    return getWidth() == getHeight();
  }

  default boolean isValidCoordinate(int w, int h) {
    return w >= 0 && w < getWidth() && h >= 0 && h < getHeight();
  }
}
```

Classe abstrata versus interface

Uma classe abstrata sem atributos e construtor, contendo apenas métodos abstratos, é na prática uma interface. A diferença técnica reside apenas na nuance de não ser permitido derivar de mais do que uma classe, ao passo que é permitido implementar várias interfaces.



Quando duas classes têm características em comum, mas conceptualmente encaixam melhor noutra hierarquia, as interfaces são o mecanismo adequado para representar essas características. Por exemplo, a interface acima poderia ser implementada por uma classe Matrix e pela classe Rectangle. Ambas consistem numa estrutura retangular (com altura e largura), que pode ser quadrada e onde faz sentido utilizar coordenadas para manipular os seus objetos. Porém, as classes não têm atributos comuns. As implementações por omissão da interface são herdadas, sendo as operações getWidth() e getHeight() definidas de forma distinta nas duas classes de acordo com os seus atributos.