# Capítulo 3 - Herança

# 1. Herança: "Filho de peixe, peixinho é"

## 1.1 O que é Herança?

Você já deve ter o conhecimento de que o encapsulamento, permite escrever objetos bem definidos e independentes. Esse recurso também permite que um objeto *use* outro objeto, através de mensagens. O *uso de trocas de mensagens* é apenas uma das maneiras pelas quais os objetos podem se relacionar na POO. A POO também fornece uma segunda maneira de relacionamento entre os objetos: herança.

A herança permite que você baseie a definição de uma nova classe a partir de uma classe previamente existente. Quando você baseia uma classe em outra, a definição da nova classe herda automaticamente todos os atributos, comportamentos e implementações presentes na classe previamente existente. Quando uma classe herda outra, todos os métodos e atributos que aparecem na interface da classe previamente existente, aparecerão automaticamente na interface da nova classe.

**Actionscript**

```
class Employee {
	private var first_name:String;
	private var last_name:String;
	private var wage:Number;
	public function Employee(first_name:String, last_name:String,
wage:Number) {
		this.first_name = first_name;
		this.last_name = last_name;
		this.wage = wage;
	}
	public function getWage():Number {
		return wage;
	}
	public function getFirstName():String {
		return first_name;
	}
	public function getLastName():String {
		return last_name;
	}
}
```

**Java**

```
public class Employee {

	private String first_name;
	private String last_name;
	private double wage;

	public Employee( String first_name, String last_name, double wage ) {
		this.first_name = first_name;
		this.last_name  = last_name;
		this.wage = wage;
	}

	public double getWage() {
		return wage;
	}

	public String getFirstName() {
```

```
            return first_name;
        }

        public String getLastName() {
            return last_name;
        }
    }
```

Instâncias de uma classe como *Employee* podem aparecer em um aplicativo de banco de dados de folha de pagamento. Agora, suponha que você precisasse modelar um funcionário comissionado. Um funcionário comissionado tem um salário-base, mais uma pequena comissão por venda. Além desse requisito simples, a classe *CommissionedEmployee* é exatamente igual à classe *Employee*. Afinal, um objeto *CommissionedEmployee* é um *Employee*.

Usando-se o encapsulamento direto, existem duas maneiras de escrever a nova classe *CommissionedEmployee*. Você poderia simplesmente repetir o código encontrado em *Employee* e adicionar o código necessário para controlar comissões e calcular o pagamento. Entretanto, se você fizer isso, terá de manter duas bases de código separadas, mas semelhantes. Se você precisar corrigir um erro, terá de fazê-lo em cada lugar. Dessa forma, simplesmente copiar e colar o código não é uma boa opção. Você precisará tentar outra coisa. Você poderia ter uma variável employee dentro da classe *CommissionedEmployee* e delegar todas as mensagens, como *getWage() e getFirstName()*, à instância de *Employee*.

*Delegação* é o processo de um objeto passar uma mensagem para outro objeto, para atender algum pedido.

Entretanto, a delegação ainda o obriga a redefinir todos os métodos encontrados na interface de *Employee* para passar todas as mensagens. Assim, nenhuma dessas duas opções parece satisfatória.

Vamos ver como a herança pode corrigir esse problema:

**Actionscript**

```
class CommissionedEmployee extends Employee {

 private var commission:Number;
private var units:Number = 0;

 public function CommissionedEmployee(first_name:String, last_name:String,
wage:Number, commission:Number) {
        super(first_name, last_name, wage);
        this.commission = commission;
}

public function calculatePay():Number {
        return getWage()+(commission*units);
}

public function addSales(units:Number) {
        this.units = this.units+units;
}

public function resetSales() {
        units = 0;
}

}
```

**Java**

```
public class CommissionedEmployee extends Employee {

      private double commission;
      private int    units;
      public CommissionedEmployee( String first_name, String last_name,
                                   double wage, double commission )
      {
            super( first_name, last_name, wage );
            this.commission   = commission;
      }

      public double calculatePay() {
            return getWage() + ( commission * units );
      }

      public void addSales( int units ) {
            this.units = this.units + units;
      }

      public void resetSales() {
            units = 0;
      }
}
```

Dessa forma, *CommissionedEmployee* baseia sua definição na classe *Employee* já existente. Como *CommissionedEmployee* herda de *Employee*, *getLastName()*, *getWage()*, *first_name, last_name* e *wage* se tornarão todos parte de sua definição. Embora as variáveis internas privadas (private) de *Employee* não podem ser vistas pelas suas filhas.

Como a interface pública de *Employee* torna-se parte da interface pública de *CommissionedEmployee*, você pode enviar para *CommissionedEmployee* qualquer mensagem que poderia enviar para *Employee*. Considere o código executor a seguir.

Crie um novo documento Flash na mesma pasta onde se encontra as classes e insira o código no primeiro frame.

**Actionscript**

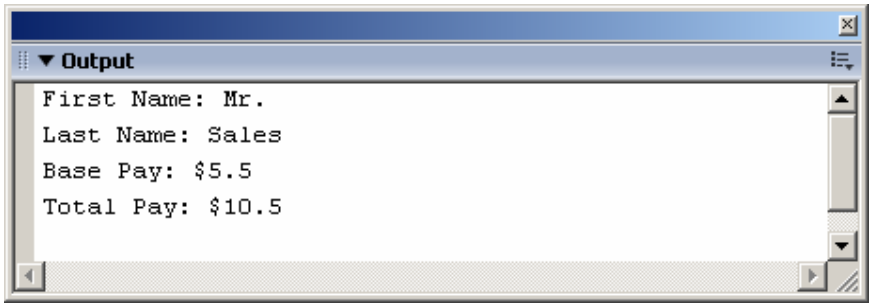
```
var c = new CommissionedEmployee("Mr.", "Sales", 5.50, 1.00);
c.addSales(5);
Imprime("First Name: "+c.getFirstName());
Imprime("Last Name: "+c.getLastName());
Imprime("Base Pay: $"+c.getWage());
Imprime("Total Pay: $"+c.calculatePay());
function Imprime(str:String) {
       trace(str);
}
```

**Java**

```
public class EmployeeExample {
      public static void main( String [] args ) {
        CommissionedEmployee c = new
CommissionedEmployee("Mr.","Sales",5.50,1.00);
        c.addSales(5);
        Print( "First Name: " + c.getFirstName() );
        Print( "Last Name: "  + c.getLastName()  );
```

```
        Print( "Base Pay: $" + c.getWage() );
        Print( "Total Pay: $" + c.calculatePay() );
      }
      public static void Print(String str)
      {
         Print(str);
      }
}
```

A execução do código acima levaria a esse resultado.




## 1.2 Refinando o conceito de herança

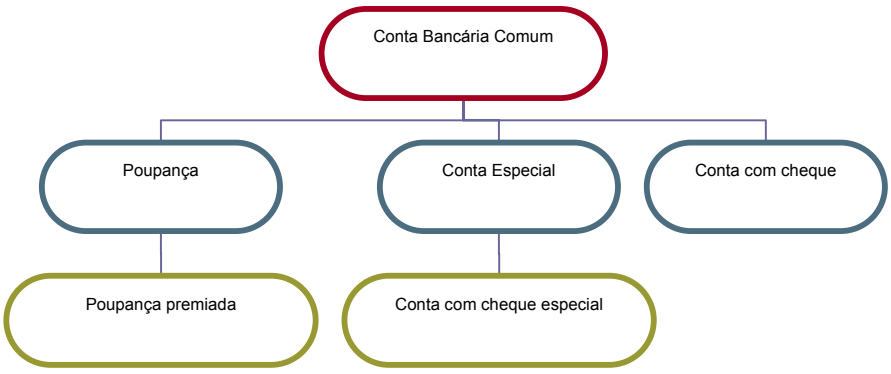
Herança é um mecanismo que permite estabelecer relacionamentos 'é um' entre classes. Esse relacionamento também permite que uma subclasse herde os atributos e comportamentos de sua classe base.

Quando uma classe filha herda uma classe progenitora, a filha obterá todos os atributos e comportamentos que a progenitora possa ter herdado de outra classe.

A classe filha sempre pode ter mais funcionalidades do que a classe mãe. Mas não o contrário.

Se você verificar que uma filha precisa remover alguma funcionalidade, isso será uma indicação de que ela deve aparecer antes da progenitora na hierarquia de herança.

Uma classe pode ter apenas uma progenitora. Assim como nós só podemos ter uma mãe. Porém, existem linguagens que permitem a herança múltipla, não é o caso do Flash.




*Interface* é uma assinatura de métodos. Algumas linguagens permitem uma progenitora e múltiplas interfaces – você é obrigado a implementar os métodos, mas não herda suas funcionalidades.

## 1.3 Atributos e métodos da classe filha

Quando uma classe herda outra, ela herda a implementação, o comportamento e os atributos. Isso significa que todos os métodos e atributos disponíveis na interface da progenitora aparecerão na interface da filha. Uma classe construída através de herança pode ter os seguintes tipos importantes de métodos e atributos:

1. Sobreposto

   A nova classe herda o método da progenitora, mas fornece uma nova definição.

2. Novo

   A nova classe adiciona um método ou atributo completamente novo.

3. Recursivo

   A nova classe simplesmente herda um método ou atributo da progenitora.

Vamos considerar um exemplo. Em seguida, exploraremos cada tipo de método e atributo.

As duas próximas classes a serem mostradas usam o princípio básico para criação de formas 2D e 3D, os métodos não estão implementados, porém, a partir desse ponto podemos ter uma idéia de como é criar objetos 3D no flash com actionscript, objetos que são criados dinamicamente por fórmulas matemáticas pré-definidas em classes.

**Actionscript**

```
class TwoDimensionalPoint
{
	private var x_coord:Number;
	private var y_coord:Number;

	public function TwoDimensionalPoint( x:Number, y:Number )
	{
		setXCoordinate( x );
		setYCoordinate( y );
	}

	public function getXCoordinate():Number
	{
		return x_coord;
	}

	public function setXCoordinate( x:Number )
	{
		x_coord = x;
	}

	public function getYCoordinate():Number
	{
		return y_coord;
	}

	public function setYCoordinate( y:Number )
	{
		y_coord = y;
	}
```

```
        public function toString() :String
        {
                return "I am a 2 dimensional point.\n" +
                        "My x coordinate is: " + getXCoordinate() + "\n" +
                        "My y coordinate is: " + getYCoordinate();
        }
}
```

**Java**

```
public class TwoDimensionalPoint
{
        private double x_coord;
        private double y_coord;

        public TwoDimensionalPoint( double x, double y )
        {
                setXCoordinate( x );
                setYCoordinate( y );
        }

        public double getXCoordinate()
        {
                return x_coord;
        }

        public void setXCoordinate( double x )
        {
                x_coord = x;
        }

        public double getYCoordinate()
        {
                return y_coord;
        }

        public void setYCoordinate( double y )
        {
                y_coord = y;
        }

        public String toString()
        {
                return "I am a 2 dimensional point.\n" +
                        "My x coordinate is: " + getXCoordinate() + "\n" +
                        "My y coordinate is: " + getYCoordinate();
        }
}
```

**Actionscript**

```
class ThreeDimensionalPoint extends TwoDimensionalPoint {

        private var z_coord:Number;

        public function ThreeDimensionalPoint(  x:Number,  y:Number,  z:Number )
{
                //chama o construtor da classe pai (TwoDimensionalPoint)
                super( x, y );
                setZCoordinate( z );
        }
```

```
        public function getZCoordinate():Number {
              return z_coord;
        }

        public function setZCoordinate( z:Number ) {
              z_coord = z;
        }

        public function toString():String {
              return "I am a 3 dimensional point.\n" +
                     "My x coordinate is: " + getXCoordinate() + "\n" +
                     "My y coordinate is: " + getYCoordinate() + "\n" +
                     "My z coordinate is: " + getZCoordinate();
        }

  }
```

**Java**

```
  public class ThreeDimensionalPoint extends TwoDimensionalPoint {

        private double z_coord;

        public ThreeDimensionalPoint( double x, double y, double z ) {
              super( x, y );
              setZCoordinate( z );
        }

        public double getZCoordinate() {
              return z_coord;
        }

        public void setZCoordinate( double z ) {
              z_coord = z;
        }

        public String toString() {
              return "I am a 3 dimensional point.\n" +
                     "My x coordinate is: " + getXCoordinate() + "\n" +
                     "My y coordinate is: " + getYCoordinate() + "\n" +
                     "My z coordinate is: " + getZCoordinate();
        }

  }
```

Aqui, você tem duas classes 'ponto' que representam pontos geométricos. Você poderia usar pontos em uma ferramenta para traçar gráficos, em um modelador virtual ou em um planejador de vôo. Os pontos têm muitos usos práticos.

*TwoDimensionalPoint* contém coordenadas x e y. A classe define métodos para obter e configurar os pontos, assim como para criar uma representação de String da instância do ponto.

*ThreeDimensionalPoint* herda de *TwoDimensionalPoint*. *ThreeDimensionalPoint* acrescenta a coordenada z, assim como um método para recuperar o valor e para configurar o valor. A classe também fornece um método para obter uma representação de String da instância. Como *ThreeDimensionalPoint* herda a classe *TwoDimensionalPoint*, ela também tem os métodos contidos dentro de *TwoDimensionalPoint*.

Esse exemplo demonstra cada tipo de método.

## Métodos e atributos sobrepostos

A herança permite que você pegue um método ou atributo previamente existente e o redefina. Dessa forma você muda o comportamento do objeto para esse método.

Um método ou atributo sobreposto aparecerá na progenitora e na filha. Por exemplo, *ThreeDimentionalPoint* redefine o método *toString()* que aparece em *TwoDimensionalPoint*.

*TwoDimensionalPoint* define um método *toString()* que identifica a instância como um ponto bidimensional e imprime suas duas coordenadas.

*ThreeDimentionalPoint* redefine o método *toString()* para identificar a instância como um ponto tridimensional e imprime suas três coordenadas.

Considere o código abaixo inserido no primeiro frame de um novo documento Flash (FLA)

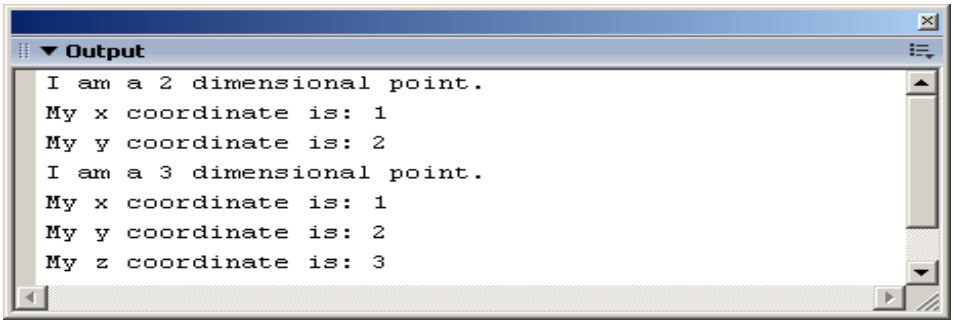
**Actionscript**

```
var two = new TwoDimensionalPoint(1,2);
var three = new ThreeDimensionalPoint(1,2,3);
trace(two.toString());
trace(three.toString());
```

**Java**

```
public static void main( String [] args ) {
   TwoDimensionalPoint two = new TwoDimensionalPoint(1,2);
   ThreeDimensionalPoint three = new ThreeDimensionalPoint(1,2,3);
   Print(two.toString());
   Print(three.toString());
}
```

A figura abaixo ilustra o que você verá após testar o documento Flash.




Sobrepor um método também é conhecido como *redefinir* um método. Redefinindo um método, a filha fornece sua própria implementação ao método, fornecendo um novo comportamento a ele. Aqui, *ThreeDimensionalPoint* redefine o comportamento do método *toString()*, para que ele seja corretamente transformado em um objeto String.

### Novos métodos e atributos

Um novo método ou atributo é um método ou atributo que aparece na filha, mas não aparece na progenitora, ou seja, a classe filha acrescenta novos métodos ou atributos em sua interface. No exemplo da classe *ThreeDimensionalPoint*, são escritos dois novos métodos: *getZCoordinate()* e *setZCoordinate()*. Com isso, é possível adicionar novas funcionalidades à interface filha.

### Métodos e atributos recursivos

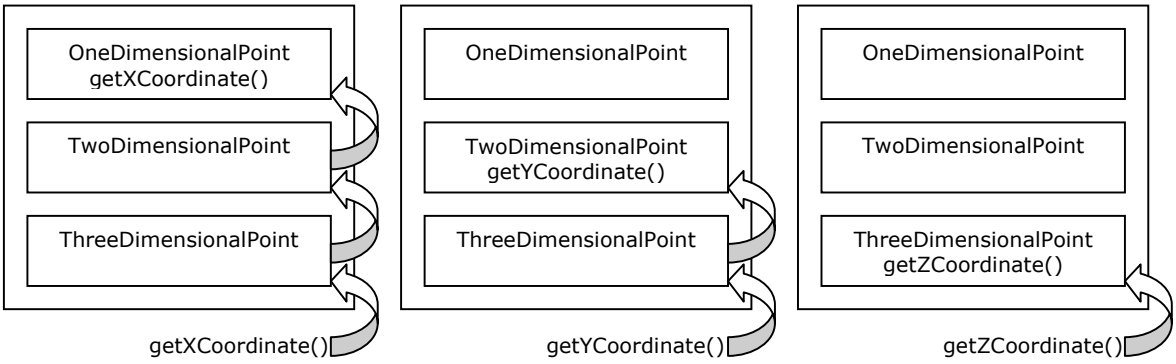
Um método ou atributo recursivo é definido na progenitora ou em alguma outra ancestral, mas não na filha. Quando você acessa o método ou atributo, a mensagem é enviada para cima na hierarquia, até que uma definição do método seja encontrada.

Você viu métodos recursivos no código-fonte de *TwoDimensionalPoint* e *ThreeDimensionalPoint*. *getXCoordinate()* é um exemplo de método recursivo, pois é definido por *TwoDimensionalPoint* e não por *ThreeDimensionalPoint*.

*Os métodos sobrepostos também podem se comportar de forma recursiva. Embora um método sobreposto apareça na filha, a maioria das linguagens orientadas a objetos fornece um mecanismo que permite a um método sobreposto chamar a versão da progenitora (ou de algum outro ancestral) do método. Essa capacidade permite que você enfatize a versão da superclasse, enquanto define novo comportamento na subclasse. Na linguagem actionscript e Java, a palavra-chave 'super' fornece acesso à implementação de uma progenitora..*

### Exemplo de mecanismo de chamada de método

A seguir temos uma ilustração de método entre os objetos ponto para uma chamada de *getXCoordinate()*. Uma chamada para *getXCoordinate()* percorrerá a hierarquia até encontrar uma definição para o método.




## 1.4 Tipos de herança

Existem três maneiras principais de usar herança:

1. Herança para reutilização de implementação
2. Herança para diferença
3. Herança para substituição de tipo

Existem alguns tipos de reutilização que são mais desejáveis que outros. Vamos ver o porquê a seguir.

## Herança para reutilização de implementação

O objetivo é reutilizar código, ou seja, a classe filha já nasce com as funcionalidades de sua progenitora.

O problema aqui é que você estará preso à implementação herdada.

**Cuidado!** Uma herança pobre é o monstro de Frankenstein da programação. Quando você usa herança unicamente para reaproveitar código, sem outras considerações, freqüentemente pode acabar com um monstro construído a partir de partes que não se encaixam, pois se lembre de que a classe está presa à implementação que herda.

## Herança para diferença

Programação por diferença significa herdar uma classe e adicionar apenas o código que torne a nova classe diferente da classe herdada.

Exemplos disso são dados pelas classes *TwoDimensionalPoint* e *ThreeDimensionalPoint* citadas anteriormente.

A programação é baseada em incrementos, o que leva a uma *especialização*.

*Especialização* é o processo de uma classe filha ser projetada em termos de como ela é diferente de sua progenitora.

A definição de classe da filha incluirá apenas os elementos que a tornam diferentes de sua progenitora. A especialização permite apenas que você adicione ou redefina os comportamentos e atributos que a filha herda de sua progenitora, porém não permite que comportamentos e atributos herdados sejam removidos da classe filha. Uma classe não obtém herança seletiva. A especialização não é uma restrição de funcionalidade, mais sim uma restrição na categorização de tipo.

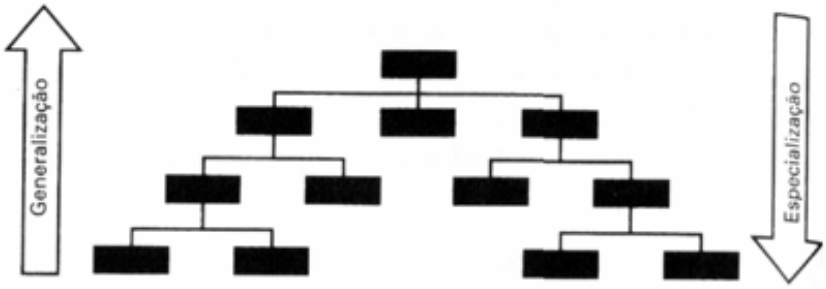
Procure não aprofundar muito hierarquicamente. Quanto menor a complexidade de seu projeto menor é o tempo de desenvolvimento, maior e mais rápida é a compreensão dele para com os desenvolvedores, mais ágil se tornará o projeto e mais manutenível também.

O código tem a tendência de se tornar menor, portanto mais gerenciável e mais rápido de ser desenvolvido.

A seguir temos alguns termos sobre classe na hierarquia:

1. Ancestral – o que antecede
2. Descendente – o que precede
3. Folha – não possui filhos

*Quando percorre uma hierarquia para cima, você generaliza. Quando percorre uma hierarquia para baixo, você especializa.*




## 1.5 Herança para substituição de tipo

A substituição de tipo permite que você descreva relacionamentos com capacidade de substituição. O que é um relacionamento com capacidade de substituição?

Considere a classe *Line*:

**Actionscript**

```
class Line
{
private var  p1:TwoDimensionalPoint;
private var  p2:TwoDimensionalPoint;

public function Line( p1:TwoDimensionalPoint , p2:TwoDimensionalPoint )
{
      this.p1 = p1;
      this.p2 = p2;
}

public function getEndpoint1() :TwoDimensionalPoint
{
      return p1;
}

public function getEndpoint2():TwoDimensionalPoint
{
      return p2;
}

public function getDistance():Number
{
  var x = Math.pow( (p2.getXCoordinate() - p1.getXCoordinate()),2);
  var y = Math.pow( (p2.getYCoordinate() - p1.getYCoordinate()),2);
  var distance = Math.sqrt( x + y );
  return distance;
}

public function  getMidpoint():TwoDimensionalPoint
{
      var new_x = ( p1.getXCoordinate() + p2.getXCoordinate() )/2;
      var new_y = ( p1.getYCoordinate() + p2.getYCoordinate() )/2;
      return new TwoDimensionalPoint( new_x, new_y );
}
}
```

**Java**

```
public class Line
{

	private TwoDimensionalPoint p1;
	private TwoDimensionalPoint p2;

	public Line( TwoDimensionalPoint p1, TwoDimensionalPoint p2 )
	{
		this.p1 = p1;
		this.p2 = p2;
	}

	public TwoDimensionalPoint getEndpoint1()
	{
		return p1;
	}

	public TwoDimensionalPoint getEndpoint2()
	{
		return p2;
	}

	public double getDistance()
	{
	  double x = Math.pow( (p2.getXCoordinate() - p1.getXCoordinate()),2);
	  double y = Math.pow( (p2.getYCoordinate() - p1.getYCoordinate()),2);
	  double distance = Math.sqrt( x + y );
	  return distance;
	}

	public TwoDimensionalPoint getMidpoint()
	{
		double new_x = ( p1.getXCoordinate() + p2.getXCoordinate() )/2;
		double new_y = ( p1.getYCoordinate() + p2.getYCoordinate() )/2;
		return new TwoDimensionalPoint( new_x, new_y );
	}
}
```

*Line* recebe dois objetos *TwoDimensionalPoint* como argumentos e fornece alguns métodos para recuperar os valores, um método para calcular a distância entre os pontos e um método para calcular o ponto médio.

Um relacionamento que tenha a capacidade de substituição permite que sejam passados para o construtor de *Line quaisquer* objetos que herdem a classe *TwoDimensionalPoint*. Lembre-se de que, quando uma filha herda sua progenitora, você diz que a filha 'é uma' progenitora. Assim como um objeto *ThreeDimensionalPoint* 'é um' objeto *TwoDimensionalPoint*, você pode passar um objeto *ThreeDimensionalPoint* para o construtor.

Considere o código abaixo digitado no primeiro frame de um novo documento Flash que se encontra na mesma pasta das classes (FLA)

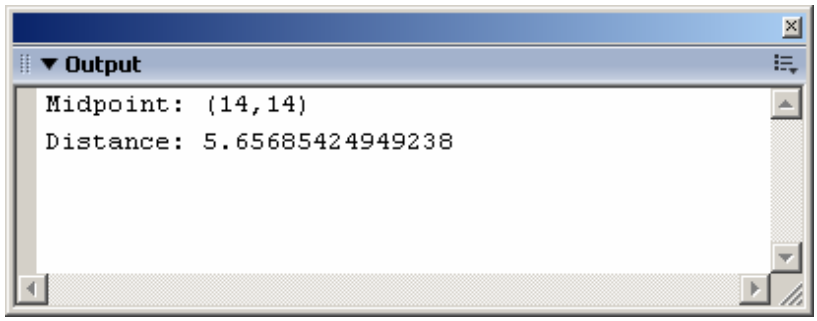
**Actionscript**

```
var p1 = new ThreeDimensionalPoint(12, 12, 2);
var p2 = new TwoDimensionalPoint(16, 16);
var l = new Line(p1, p2);
var mid = l.getMidpoint();
trace("Midpoint: ("+mid.getXCoordinate()+","+mid.getYCoordinate()+")");
trace("Distance: "+l.getDistance());
```

**Java**

```
public static void main( String [] args ) {
        ThreeDimensionalPoint p1 = new ThreeDimensionalPoint( 12, 12, 2 );
        TwoDimensionalPoint p2   = new TwoDimensionalPoint( 16, 16 );
        Line l = new Line( p1, p2 );
        TwoDimensionalPoint mid = l.getMidpoint();
        System.out.println( "Midpoint: (" +
                    mid.getXCoordinate() + "," + mid.getYCoordinate() + ")" );
        System.out.println( "Distance: " + l.getDistance() );
}
```

Você notará que o método principal passa um objeto *TwoDimensionalPoint* e um objeto *ThreeDimensionalPoint* para o construtor de *Line*. A figura abaixo ilustra o que aparecerá ao testar o documento Flash.




Capacidade de conexão é um conceito poderoso. Como você pode enviar a uma filha qualquer mensagem que pode ser enviada para sua progenitora, é possível tratá-la como se ela pudesse ser substituída pela progenitora. Esse é o motivo pelo qual você não deve *remover* comportamentos ao criar uma filha. Se você fizer isso, a capacidade de conexão será invalidada.

Um *subtipo* é um tipo que estende outro tipo através de herança.

A capacidade de substituição é importante, pois ela permite que você escreva código genérico. Em vez de ter várias instruções "case" ou testes "if/else" para ver que tipo de ponto o programa estava usando, você simplesmente programa seus objetos para tratar com objetos do tipo *TwoDimensionalPoint*.

## 1.6 Como a herança atende os objetivos da Orientação a Objetos

A herança preenche cada um dos objetivos da POO. Ela ajuda a produzir software que é:

1. Natural
2. Confiável
3. Reutilizável
4. Manutenível
5. Extensível

6. Oportuno

Ela atinge esses objetivos, como segue:

- Natural

  A herança permite que você modele o mundo mais naturalmente. Através da herança, você pode formar hierarquias de relacionamento complexas entre suas classes. Como seres humanos, nossa tendência natural é querer categorizar e agrupar os objetos que estão em torno de nós. A herança permite que você traga essas tendências para a programação.

  A herança também abrange o desejo do programador de evitar trabalho repetitivo.

- Confiável

  A herança simplifica o código. Quando você programa pela diferença, você adiciona apenas o código que descreve a diferença entre a progenitora e a filha. Como resultado, cada classe pode ter código menor. Cada classe pode ser especializada no que faz. Menos código significa menos erros.

  A herança permite a reutilização de código bem testado e comprovado, como a base de suas novas classes. A reutilização de código comprovado é sempre mais desejável do que escrever novo código.

  Finalmente, o mecanismo de herança em si é confiável. O mecanismo é incorporado à linguagem, de modo que você não precisa construir seu próprio mecanismo de herança e certificar-se de que todo mundo segue suas regras.

- Reutilizável

  A base da herança é a reutilização, onde você pode reaproveitar classes antigas para construir novas classes.

  A herança permite também que você reutilize classes de maneira nunca imaginada pela pessoa que escreveu a classe. Sobrepondo e programando pela diferença, você pode alterar o comportamento de classes existentes e usá-las de novas maneiras.

- Manutenível

  A reutilização de código testado significa que você terá menos erros em seu novo código. E quando você encontrar um erro em uma classe, todas as subclasses tirarão proveito da correção.

  Em vez de se aprofundar no código e adicionar recursos diretamente, a herança permite que você pegue código previamente existente e o trate como a base da construção de uma nova classe. Todos os métodos, atributos e informações de tipo se tornam parte de sua nova classe. Ao contrário do recorte e colagem, existe apenas uma cópia do código original para manter. Isso ajuda na manutenção, diminuindo a quantidade de código que você precisa manter.

  Se você fosse fazer alterações diretamente no código existente, poderia danificar a classe base e afetar partes do sistema que usem essa classe.

- Extensível

  A herança torna a extensão ou especialização de classe possível. Uma classe antiga pode ter novas funcionalidades a qualquer momento, apenas pela

adição de novos métodos. A programação pela diferença e a herança para capacidade de conexão estimulam a extensão de classes.

- Oportuno

  Você já viu como a reutilização simples pode diminuir o tempo de desenvolvimento. Programar pela diferença significa que existe menos código para escrever; portanto, você deve terminar mais rapidamente. Capacidade de substituição significa que você pode adicionar novos recursos, sem ter de alterar muito o código já existente.

  A herança também pode tornar os testes mais fáceis, pois é necessário apenas testar a nova funcionalidade e qualquer interação com a funcionalidade antiga.

## 1.7 Perguntas – Exercícios

1) Quais são algumas das limitações da reutilização simples?

2) O que é herança?

3) Quais são as três formas de herança?

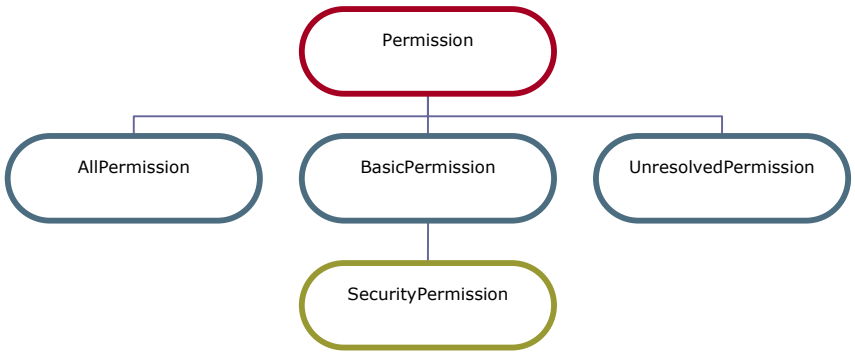
4) Por que a herança de implementação é perigosa?

5) O que é programação pela diferença?

6) Ao herdar uma classe, pode-se ter três tipos de métodos e atributos. Quais são esses três tipos de atributos e métodos?

7) Quais vantagens a programação pela diferença oferece?

8) Considere a hierarquia abaixo:



a. Se você voltar sua atenção para a classe *Permission*, quais classes são suas filhas? Quais são descendentes?

b. Considerando a hierarquia inteira, qual classe é a classe raiz? Quais classes são classes folhas?

c. *Permission* é uma ancestral de *SecurityPermission*?

9) O que é herança para substituição de tipo?

Ao terminar de responder as questões acima, envie-as em um arquivo texto ao seu tutor.