Interfaces

Sumario

Interfaces

Hai un número de situacións dentro do desenvolvemento de software onde grupos diferentes de programadores debería cinguirse a un "contrato" que indique como o seu software interatúa. Cada grupo debería ser capaz de escribir o seu código sen ningún coñecemento do software que están a escribir outros grupos. De forma xeral, os interfaces representan eses contratos.

Como exemplo da utilización de interfaces supoñamos unha sociedade hipotética e futura onde os coches se autoconducen. Os construtores de automóbiles escriben software para conducir o coche ? parar, acelerar, xirar a esquerda, xirar a dereita, etc-. Outras empresas encargaranse de conducir realmente o coche a partires dos datos GPS da súa localización, mapas, tráfico, etc. utilizando o anterior software.

Os construtores deben publicar uns métodos estándar para conducir un coche indicando o xeito de realizar cada unha das operacións necesarias na condución. Os sistemas de guiado escribirán o software que utilice esta interface para conducir realmente o coche. Ningún dos grupos de constructores(os de coches e os de sistemas de guiado automático) precisa coñecer nada do software do outro grupo.

Interfaces en Java

Un interface en Java é un tipo de referencia, similar a unha clase, onde só pode haber constantes, sinaturas de métodos e tipos aniñados. Nunha interface nunca se escribirá o corpo dun método. Os interfaces non poden ser instanciados, só poden ser implementados por unha clase ou estendidos por outra interface.

Definir unha interface é similar a crear unha nova clase:

```
public interface ConducirCoche {
  // Declaración de constantes se houbese algunha
  // Sinaturas dos métodos
  int xirar(Direccion direccion,   // Enum cos valores ESQUERDA, DEREITA
            double radio, double velocidadeInicial, double velocidadeFinal);
  int cambiarCarril(Direccion  direccion, double velocidadeInicial, double velocidadeFinal);
  int sinalizarXiro(Direccion  direccion, boolean intermitentePosto);
  int getRadarFrente(double distanciaOCoche, double velocidade);
  int getRadarDetras(double distanciaOCoche, double velocidade);
  // máis sinaturas para outros métodos.
}
```

Observa que as **sinaturas dos métodos non levan chaves e rematan en punto e coma**. Para usar unha interface é preciso escribir unha clase que a implemente, é dicir que proporcione un corpo para cada un dos métodos declarados dentro da interface. Por exemplo:

```
public class ConducirSeatIbiza implements ConducirCoche {
  int sinalizarXiro(Direccion  direccion, boolean intermitentePosto) {
     //código que sinaliza o xiro dun Seat Ibiza usando os intermitentes.
  }
  // etc...
}
```

No exemplo anterior dos coches autoguiados son os constructores de coches os que implementan a interface. Por exemplo, a implementación de Seat será substancialmente distinta da de Nissan, pero ambos teñen que se cinguir o que a interface di. Os construtores de sistemas de condución automática serán os clientes da interface e construirán sistemas que a partires do uso dos datos GPS da posición do coche, mapas, e datos de tráfico conducirán realmente o coche. Ó facer isto, o sistema de condución automática estará invocando cada un dos métodos da interface: xirar,

cambiarCarril, etc.

Unha interface é un API

O exemplo dos coches intelixentes mostra como unha interface é usada como unha API. As API's son moi comúns nos produtos de software comercial. Normalmente unha compañía vende paquetes de software complexos para que outra compañía os use dentro do seu propio software. Como exemplo supoñamos un paquete de software con métodos de procesamento de imaxes dixitais que será vendido a outra compañía adicada ao software gráfico para o usuario final. A compañía adicada ao procesamento de imaxes escribe as súas propias clases implementando unha interface pública para os seus clientes. A compañía de gráficos para o usuario final invoca os métodos de procesamento de imaxes utilizando as sinaturas dos mesmos e obtén os tipos de datos especificados na interface. Mentres a interface ou API da compañía adicada ao procesamento de imaxes é pública, a súa implementación gárdase en segredo. A compañía de procesamento de imaxes pode modificar ao seu antollo as clases que implementan esa interface pública mentres este non cambie.

Interfaces e herdanza múltiple

Java non permite herdanza múltiple (clases que herdan de varias clases) sen embargo as interfaces proporcionarán unha alternativa para simular esta característica.

Un clase pode herdar unicamente dunha superclase pero pode implementar máis dunha interface. Polo tanto un obxecto pode ter múltiples tipos: o tipo da súa propia clase, e o tipo de todos os interfaces que implementa. Isto significa que se declaramos unha variable de tipo interface, o seu valor poderá referenciar calquera obxecto instanciado a partires de calquera clase que implemente esa interface.

Definir un interface

Unha declaración dun interface consiste nun modificador de acceso, a palabra reservada `interface`, o nome da interface, unha lista de interfaces pais separadas por comas (caso de existir) e o corpo do interface. Por exemplo:

```
public interface InterfaceAgrupado extends Interface1, Interface2, Interface3 {
  // Declaración de constantes
  double E = 2.718282;
  // sinatura dos métodos
  void facerAlgo (int i, double x);
  int facerAlgoMais(String s);
}
```

O acceso public indica que este interface poderá ser usado por calquera clase en calquera paquete. Se non se especifica que un interface é público, este será unicamente accesible dende o paquete dentro do que está definido.

Un interface pode estender outros, do mesmo xeito que un clase pode estender outras. Sen embargo mentres unha clase pode estender soamente outra clase (superclase), un interface pode estender calquera número de interfaces pai. A declaración do interface inclúe unha lista separada por comas cos interfaces que estende.

O corpo do interface

O corpo conterá a declaración de todos os métodos incluídos dentro do interface. Cada declaración remata con punto e coma, pero non leva chaves xa que o interface non conterá as implementacións dos métodos declarados dentro del. Todos os métodos declarados dentro dun interface son implicitamente public, de xeito que o modificador pode ser omitido.

Un interface pode conter, ademais de sinaturas de métodos, declaracións de constantes que serán implicitamente public, static e final. De novo estes modificadores poderán ser omitidos.

Implementar un interface

Para declarar unha clase que implemente un interface débese incluír a palabra reservada `implements` na declaración da clase. Unha clase pode implementar calquera número de interfaces, deste xeito, implements irá seguida dunha lista separada por comas con todos os interfaces que hai que implementar.

Por convención, a palabra `implements` irá a continuación de `extends`, caso de que esta última exista.

Considera un interface que define o xeito de comparar o tamaño de dous obxectos:

```
public interface IRelacionable {
  // Devolverá 1, 0, -1 se este é maior, igual
  // ou menor que o outro
  public int eMaiorQue( IRelacionable outro);
}
```

Se queremos comparar o tamaño de dous obxectos similares, sen importar de que tipo sexan, a clase que os instancie debería implementar *IRelacionable*.

Calquera clase pode implementar *IRelacionable* sempre e cando haxa algunha maneira de comparar o tamaño dos obxectos instanciados por esa clase. Así, para un *String* podería ser o número de caracteres, para os libros o número de páxinas, para os estudantes o seu peso, etc. Para os obxectos xeométricos planos o área é unha boa elección, mentres que o volume sería axeitado para os obxectos en 3 dimensións. Todos estas clases poden implementar un método *eMaiorQue()*.

Se sabemos que unha determinada clase implementa *IRelacionable*, entón sabemos que os obxectos instanciados por esa clase poden ser comparados.

A continuación mostrase a clase *RectanguloPlus*, sendo esta unha clase Rectangulo(vista anterioremente) que implementa *IRelacionable*.

```
public class RectanguloPlus implements IRelacionable {
   public int ancho = 0;
   public int alto = 0;
   public Punto orixe;
   // 4 construtores
   public RectanguloPlus() {
       orixe = new Punto(0, 0);
   }
   public RectanguloPlus(Punto p) {
       orixe = p;
   }
   public RectanguloPlus(int w, int h) {
       orixe = new Punto(0, 0);
       ancho = w;
       alto = h;
   }
   public RectanguloPlus(Punto p, int w, int h) {
       orixe = p;
       ancho = w;
       alto = h;
   }
   // un método para mover o rectángulo
   public void mover(int x, int y) {
       orixe.x = x;
       orixe.y = y;
   }
   // Un método para calcular o área do rectángulo
   public int obterArea() {
       return ancho * alto;
   }
   // o método que implementa IRelacionable
   public int eMaiorQue(IRelacionable outro) {
       RectanguloPlus outroRect = (RectanguloPlus)outro;
       if (this.obterArea () < outroRect.obterArea ())
               return -1;
       else if (this.obterArea () > outroRect.obterArea ())
               return 1;
       else
               return 0;
   }
}
```

Xa que *RectanguloPlus* implementa *IRelatable*, o tamaño de dous obxectos calquera pertencentes á clase *RectanguloPlus* poderá ser comparado.

Usar interfaces como tipos

Cando se define un novo interface, estase a definir unha referencia a un tipo de datos. Os nomes dos interfaces poden ser utilizados como calquera outro tipo de datos. Cando se define unha variable de tipo interface, calquera obxecto que se lle asigne debe ser unha instancia dunha clase que implemente ese interface.

Como exemplo, a continuación móstrase un método para atopar o obxecto maior de entre un par que sexan instancias dunha clase que implemente *IRelacionable*:

```
public Object atoparMaior(Object obxecto1, Object obxecto2) {
  IRelacionable obx1 = (IRelacionable)obxecto1;
  IRelacionable obx2 = (IRelacionable)obxecto2;
  if ( (obx1).eMaiorQue(obx2) > 0)
     return obxecto1;
  else
     return obxecto2;
}
```

Facendo un casting de *obxecto1* ata o tipo *IRelacionable* xa podemos invocar o método *eMaiorQue()*.

Se o interface *IRelacionable* está implementado nunha ampla variedade de clases, calquera obxecto instancia de calquera desas clases poderá ser comparado usando o método *atoparMaior()*, xa que ambos obxectos son da mesma clase. Do mesmo xeito os obxectos poderán ser comparados usando os seguintes métodos:

```
public Object atoparMenor(Object obxecto1, Object obxecto2) {
  IRelacionable obx1 = (IRelacionable)obxecto1;
  IRelacionable obx2 = (IRelacionable)obxecto2;
  if ( (obx1).eMaiorQue(obx2) < 0)
     return obxecto1;
  else
     return obxecto2;
}
public boolean eIgual(Object obxecto1, Object obxecto2) {
  IRelacionable obx1 = (IRelacionable)obxecto1;
  IRelacionable obx2 = (IRelacionable)obxecto2;
  if ((obx1).eMaiorQue(obx2) == 0)
     return true;
  else
     return false;
}
```

Estes métodos funcionarán para calquera obxecto instancia dunha clase que implemente *IRelacionable*.

Modificar interfaces

Consideremos o seguinte interface:

```
public interface Algo {
  void facerAlgo(int i, double x);
  int faceOutraCousa(String s);
}
```

Supoñamos que máis tarde queremos engadir outro método:

```
public interface Algo {
  void facerAlgo(int i, double x);
  int faceOutraCousa(String s);
  boolean funciona(int i, double x, String s);
}
```

Se facemos este cambio todas as clases que implementen o vello interface *Algo* deixarán de funcionar xa que ningunha delas implementará o novo método *funciona()*.

Debemos tratar de anticipar todos os usos que o interface recibirá, e polo tanto escribir todos os seus métodos a primeira vez. Como isto é practicamente imposible, e as veces hai que modificar un interface, o xeito de facelo e evitar o anterior problema é crear outro interface que estenda o vello, onde poremos as sinaturas dos novos métodos. Por exemplo:

```
public interface AlgoMais extends Algo {
  boolean funciona(int i, double x, String s);
}
```

Agora os usuarios do interface poderán escoller entre seguir a utilizar o vello interface *Algo* ou actualizarse a este novo *AlgoMais*(terá os 2 métodos de

Algo máis ese novo método engadido).