Les classes : héritage et polymorphisme

Loïc Joly, Michel Simatic, Amina Guermouche

Télécom SudParis

17 Mai 2018

Héritage

1. Syntaxe : La classe Circle hérite de la classe Shape, se traduit par `class Circle : public Shape`

```
class Shape{
  private:
     int pos_x, pos_y;
  protected:
  /* liste des attributs visibles par les
      classes filles*/
  public:
  /* liste des fonctions */
}

class Circle : public Shape {
  private:
     int rayon;
  public:
  /* liste des fonctions */
  }
```

Copie des objets et héritage (1/2)

1. Lorsque l'on veut copier un objet de type classe Circle (classe fille) dans un objet de type Shape (classe mère), un phénomène appelé *"slicing"* se produit : seuls les champs définis dans la classe mère sont gardés dans l'objet créés/

```
class Shape{
  int x, y;
};

class Circle : public Shape{
  int rayon;
};
int main()
{
  Circle c{...}; /* appel au constructeur*/
  Shape s = c; /* s ne va stocker que les
      elements de Shape (x et y) : slicing*/
}
```

Copie des objets et héritage (2/2)

2. Afin d'éviter le problème de slicing, on utilisera des pointeurs vers des objets

```
class Shape{
  int x, y;
};

class Circle : public Shape{
  int rayon;
};
int main()
{
  vector<Shape*> v;
  Circle *c = new Circle(...);
  v.push_back(c); /* Meme si v contient des
      Shape*, le pointeur pointera sur un
      objet de type Circle*/
}
```

Fonctions virtuelles (1/2)

1. Une méthode virtuelle est définie dans la classe mère et dans au moins une classe fille. La signature des fonctions doit être la même pour toutes les redéfinitions. Au niveau des classes filles, on parlera d'*override*
2. Déclaration

```
class Shape{
  virtual void print(); /* la fonction est
      definie dans le .cpp correspondant*/
};

class Circle : public Shape{
  void print() override;
};
```

3. Le mot clé *override* n'est pas obligatoire. Il est cependant recommandé car il permet au compilateur de nous signaler des erreurs lors de la redéfininion de la fonction (paramètres différents, type de retour différent, ...).

Fonctions virtuelles (2/2)

1. Grâce au mot clé virtual, le compilateur va regarder le type de l'objet appelant la méthode afin d'utiliser la bonne fonction.

```
class Shape{
  virtual void print(); /* la fonction est
      definie dans le .cpp correspondant*/
};
class Circle : public Shape{
  void print() override;
};
int main()
{
  vector<Shape*> s;
  Circle *c = new Circle(...);
  s.push_back(c);
  s[0]->print(); /* appel a la fonction print
      () de la classe Circle car la premiere
      case de v contient un pointeur vers un
      objet de type Circle*/
}
```

Fonction virtuelles pures (1/2)

1. Parfois, la définition d'une méthode virtuelle dans la classe mère n'a pas de sens. Dans ce cas, on parlera de méthode virtuelle pure.

```
class Shape{
  virtual void print() = 0; /* ici print est
      une fonction virtuelle pure*/
};

class Circle : public Shape{
  void print() override;
};
```

2. Dès qu'une classe contient une fonction virtuelle pure, elle devient abstraite. On ne peut donc plus l'instancier.

Fonction virtuelles pures (2/2)

2. Remarque 1 : Toutes les classes héritant d'une classe abstraite doivent implémenter les fonctions virtuelles pures de la classe mère (sinon elles aussi seront abstraites).

3. Remarque 2 : il est possible de définir la fonction virtuelle pure dans la classe mère. L'intérêt de cette fonction est de rendre la classe abstraite et donc non instanciable.

4. Remarque 3 : On peut appeler le constructeur d'une classe abstraite dans une classe fille. De ce fait, le constructeur d'une classe abstraite doit être protected étant donné que seules ses classes filles vont l'appeler.