Master 2 LITL
Programmation pour le TAL (SLT0905V)
Programmation orientée objets : notions

Franck Sajous/CLLE-ERSS

Séance 2

http://fsajous.free.fr/

POO : principes

Propriétés

- maintient les principes de la programmation impérative structurée
- introduit la notion d'objets, de classes, d'encapsulation, d'héritage...
- facilite la réutilisabilité du code, le développement en équipe

Adaptation du paradigme *programme = instructions + données*

- prog. structurée : structures de données et procédures ($\approx$ fonctions)
- POO : modélisation d'objets du monde réel ou de concepts abstraits
  Objet = membres/attributs ($\approx$ données) + méthodes ($\approx$ fonctions)
- Principes fondamentaux :
  - Encapsulation : un objet manipule ses propres données uniquement.
    En POO pure, un objet X n'a accès aux données d'un objet Y qu'à travers les méthodes de Y qui constituent son "interface".
  - Pas de « classe amie » (cf. C++) en Java
  - Abstraction des données (implémentation masquée) : on ne connaît pas le détail de l'implémentation de la méthode d'un objet que l'on utilise

  $\rightarrow$ exemples à venir

Notion de classe

Classes et types

- Comme dans d'autres langage, il existe des *types primitifs* : nombres entiers (`int`) ou réels (`float`), caractères (`char`), etc.
- Ces types sont *prédéfinis*. Les opérations que l'on peut effectuer sur ces types sont connues et limitées : on peut multiplier deux entiers entre eux et concaténer deux chaînes, mais pas l'inverse.
- Classe = structure de données plus complexe créée par le développeur (ou fournie par l'API) avec les méthodes en relation avec ces données.
- type primitif $\neq$ classe en Java est un demi-mensonge $\rightarrow$ clarification à venir
- Ex. : dans un prog. d'étiquetage morphosyntaxique, on veut modéliser le concept de *token*. Un token a une *forme*, un *lemme*, une *étiquette*, et éventuellement d'autres traits : une *longueur*, une position dans la phrase (*index*), une *casse* (tout en minuscules, tout en majuscules, 1ère lettre en majuscule, etc.).

Notion de classe

Classes et types

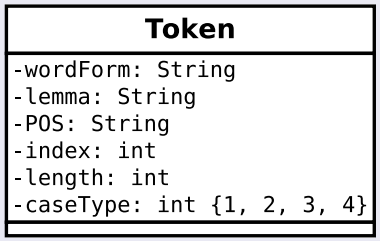
- Comme dans d'autres langage, il existe des *types primitifs* : nombres entiers (`int`) ou réels (`float`), caractères (`char`), etc.
- Ces types sont *prédéfinis*. Les opérations que l'on peut effectuer sur ces types sont connues et limitées : on peut multiplier deux entiers entre eux et concaténer deux chaînes, mais pas l'inverse.
- Classe = structure de données plus complexe créée par le développeur (ou fournie par l'API) avec les méthodes en relation avec ces données.
- type primitif $\neq$ classe en Java est un demi-mensonge $\rightarrow$ clarification à venir
- Ex. : dans un prog. d'étiquetage morphosyntaxique, on veut modéliser le concept de *token*. Un token a une *forme*, un *lemme*, une *étiquette*, et éventuellement d'autres traits : une *longueur*, une position dans la phrase (*index*), une *casse* (tout en minuscules, tout en majuscules, 1ère lettre en majuscule, etc.).

Token
-wordForm: String -lemma: String -POS: String -index: int -length: int -caseType: int {1, 2, 3, 4}

Classes et objets

Classe

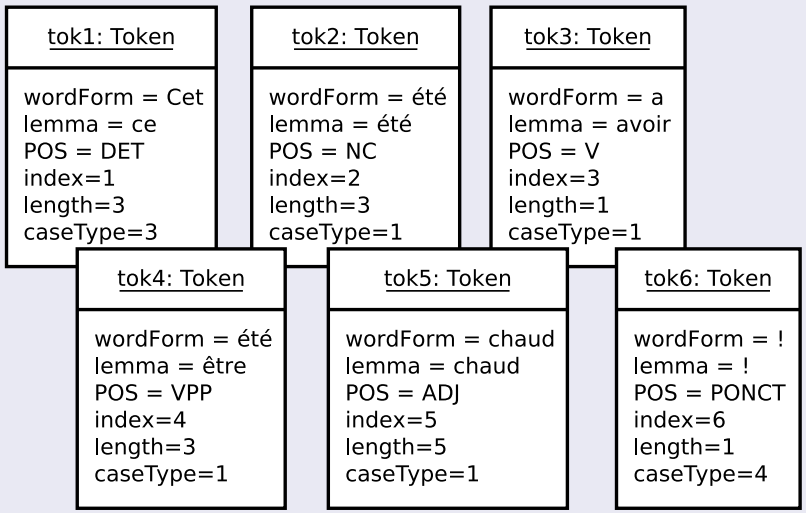
- **Classe** *token* = abstraction de ce qu'est/peut être un token donné
  = ensemble de caractéristiques (désincarnées, *i.e.* non valuées) que possèdent tous les tokens possibles



Objets

UN ETE CANICULAIRE

Cet été a été chaud !



- **Objets : instances** de classe, déclinaisons réelles
  Structure commune ($\rightarrow$ même classe), valeurs différentes des membres

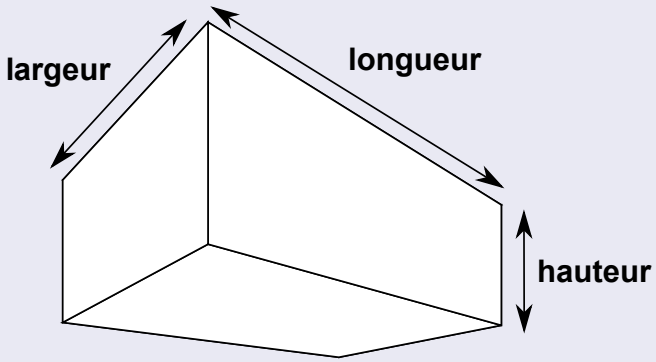
Classes et objets

Exemple : systèmes d'information (d'un université, d'une bibliothèque, etc.)

- classe *Humain* et instances (individu) *John Doo, Jean-Pierre Dupont, ...*
- classe *Etudiant* et les instances *Etu123456, Etu789123, ...*
- classe *UE* et instances *SLT0905V, SLT0906V, ...*
- classe *Livre* et les instances *"Thinking in Java"*, *"UML Distilled"*, *"The Agile Samurai"*, etc.

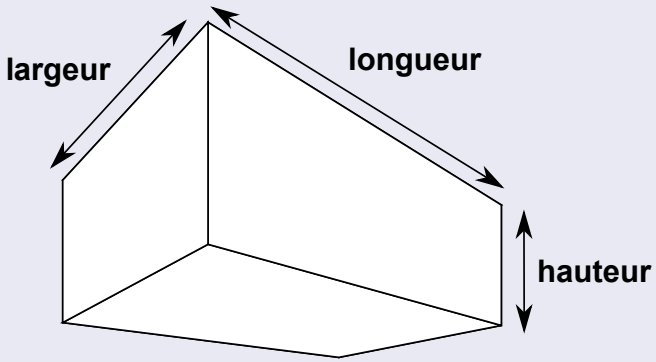
POO : classes et objets

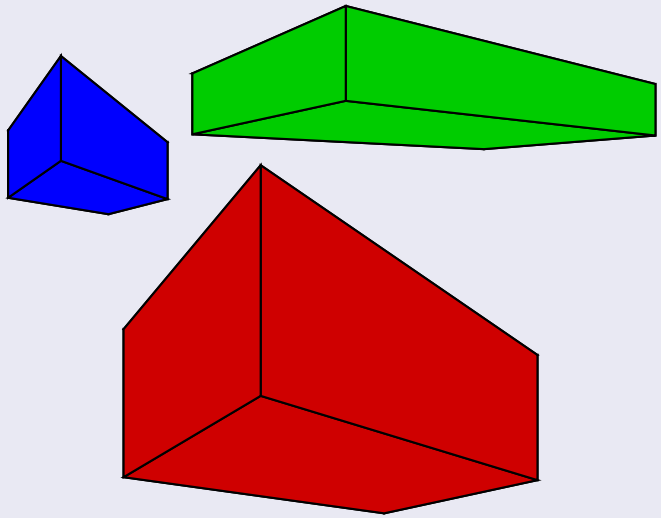
Exemple : jeu de construction en 3D



POO : classes et objets

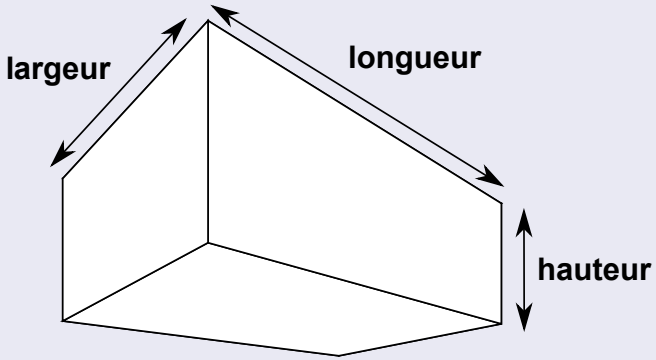
Exemple : jeu de construction en 3D

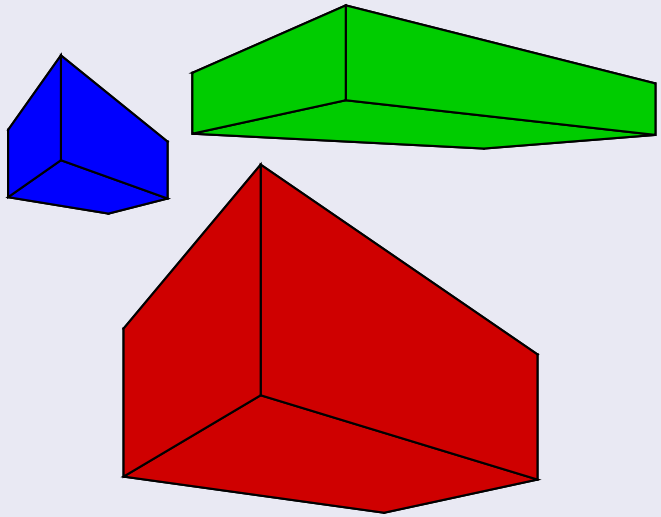




POO : classes et objets

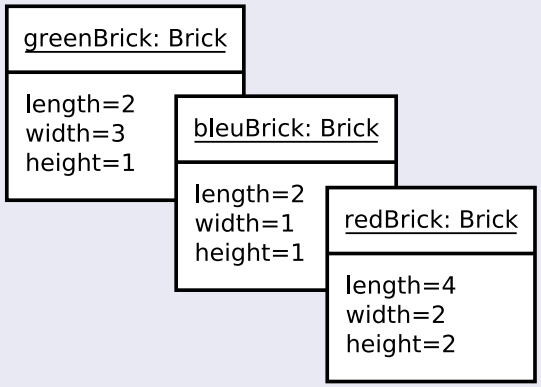
Exemple : jeu de construction en 3D





Brick

-length: int
-width: int
-height: int



UML / Java

UML

Diagramme de classes

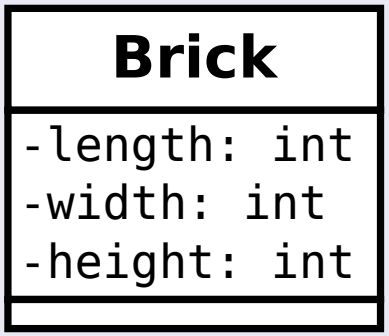
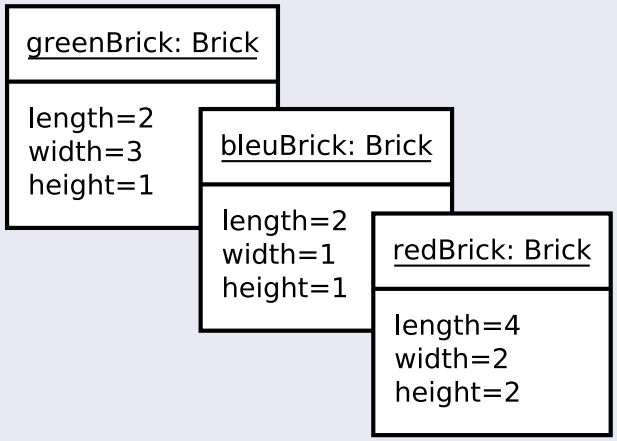


Diagramme d'objets



Java

Fichier `Brick.java`

```
public class Brick
{
    private int length;
    private int width;
    private int height;
}
```

Définition de la classe

Quelque part ailleurs (ou pas)...

```
Brick blueBrick = new Brick();
Brick greenBrick = new Brick();
Brick redBrick = new Brick();
```

Utilisation de la classe : création (instanciation) d'objets de la classe

Méthodes

Exemples précédents : classes Token et Brick

- définition d'une « structure de données »
- membres destinés à recevoir des valeurs

Méthodes

Exemples précédents : classes Token et Brick

- définition d'une « structure de données »
- membres destinés à recevoir des valeurs
- mais on n'en fait rien !
- comment affecter une valeur donnée à un membre, utiliser la valeur d'un membre ?

Méthodes

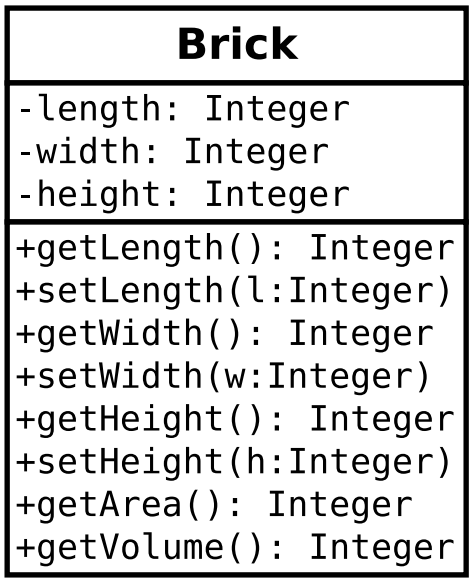
Exemples précédents : classes Token et Brick

- définition d'une « structure de données »
- membres destinés à recevoir des valeurs
- mais on n'en fait rien !
- comment affecter une valeur donnée à un membre, utiliser la valeur d'un membre ?

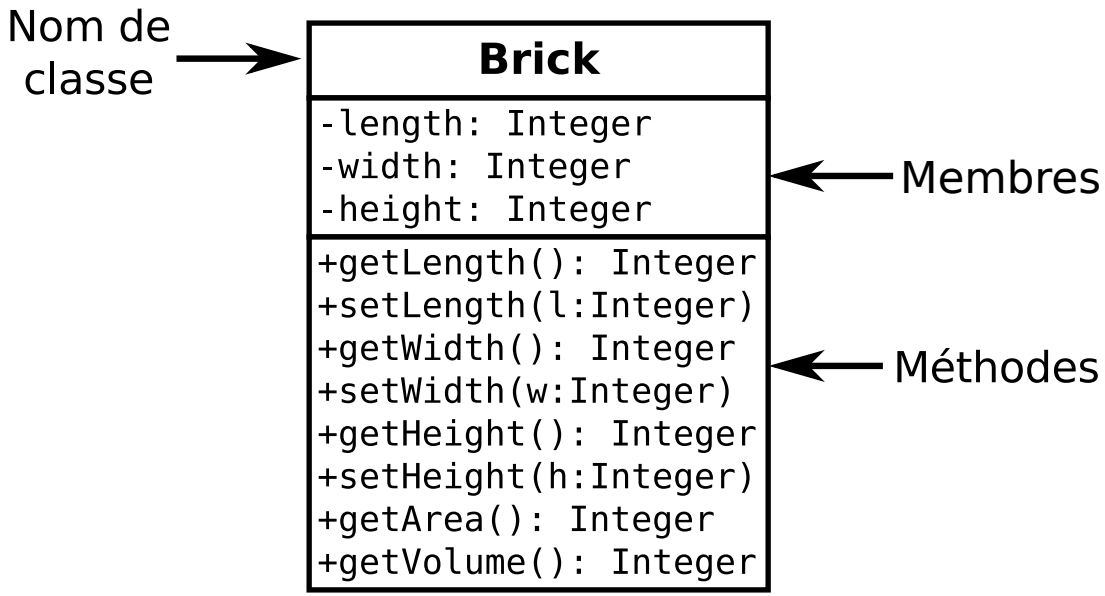
Méthodes

- ≈ procédure/fonction propre à un objet
- permettent de changer la valeur d'un membre ou d'y accéder (*getters et setters*)
- permettent de créer de nouveaux objets, d'effectuer des calculs, des entrées-sorties, etc.
- on invoque une méthode « sur » un objet : `nomObjet.nomMethode ()`
- un objet peut invoquer une de ses propres méthodes (sur lui-même) : `nomMethode ()`

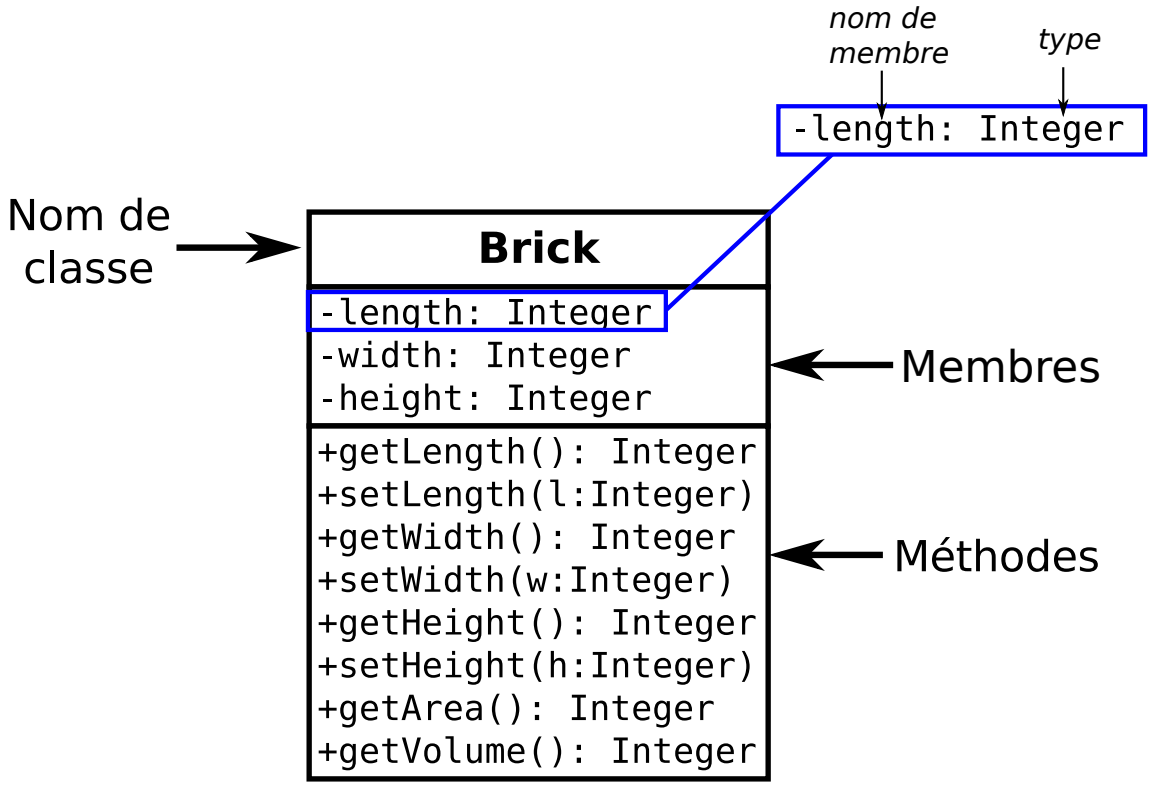
Classe, membres et méthodes : représentation UML



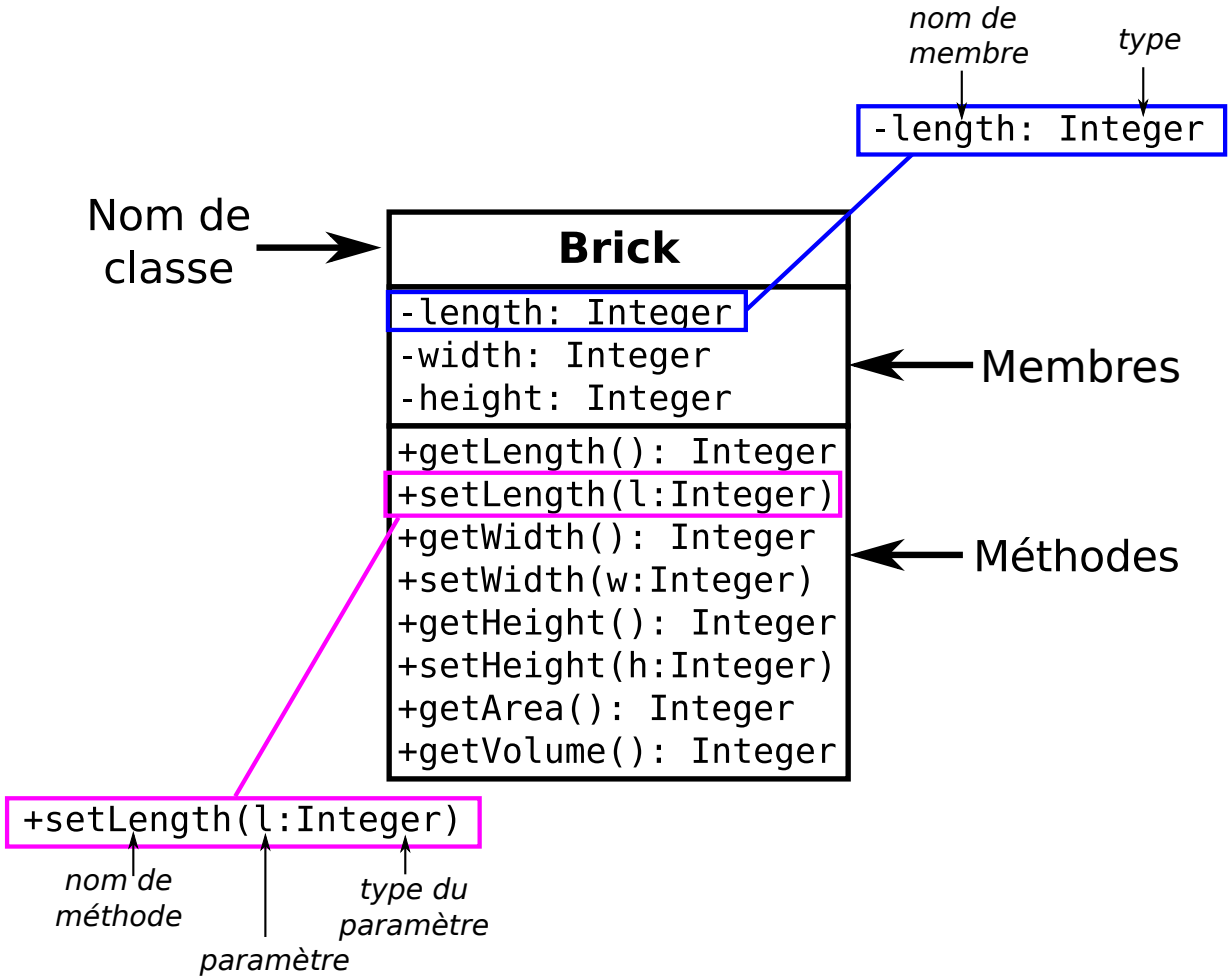
Classe, membres et méthodes : représentation UML



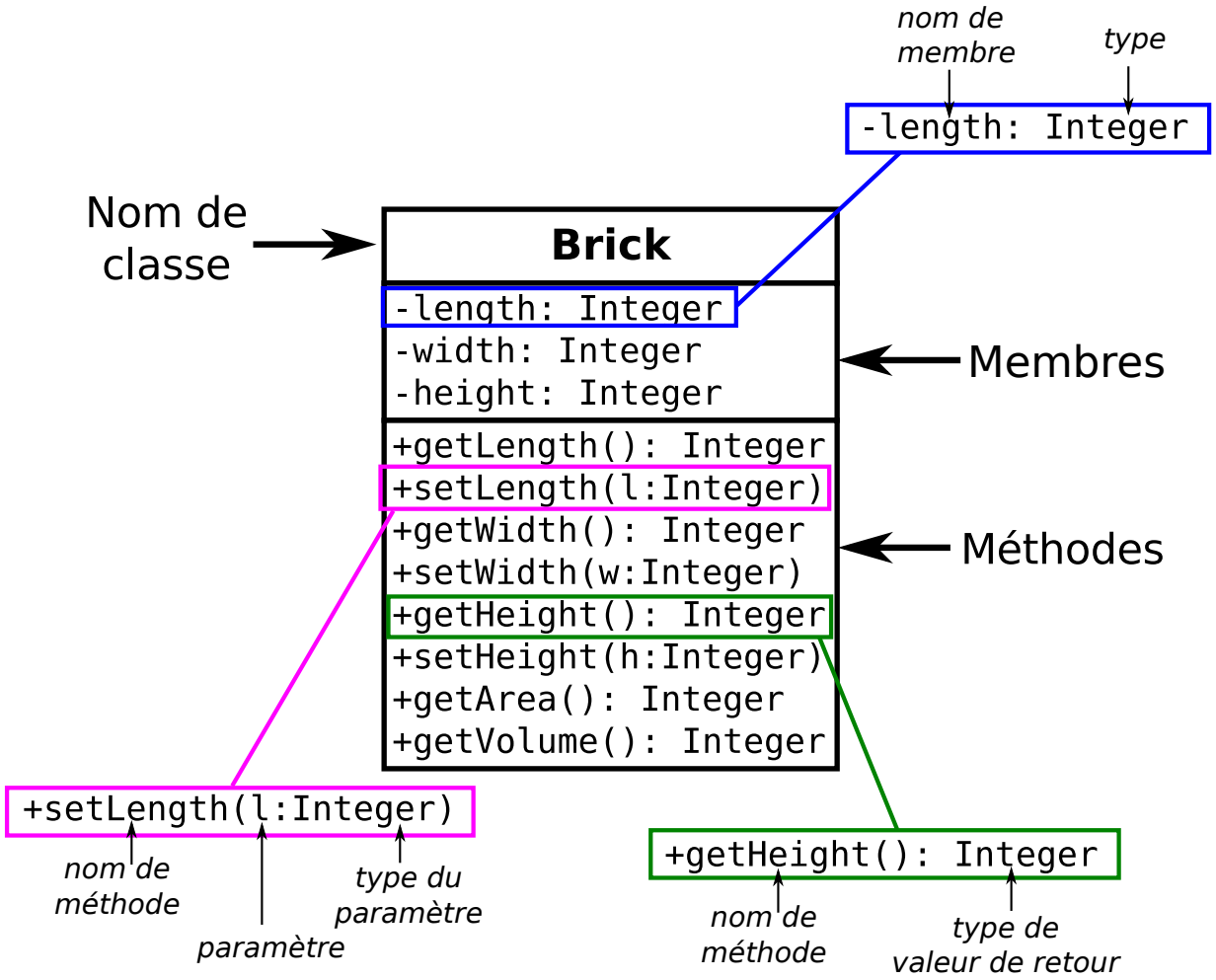
Classe, membres et méthodes : représentation UML



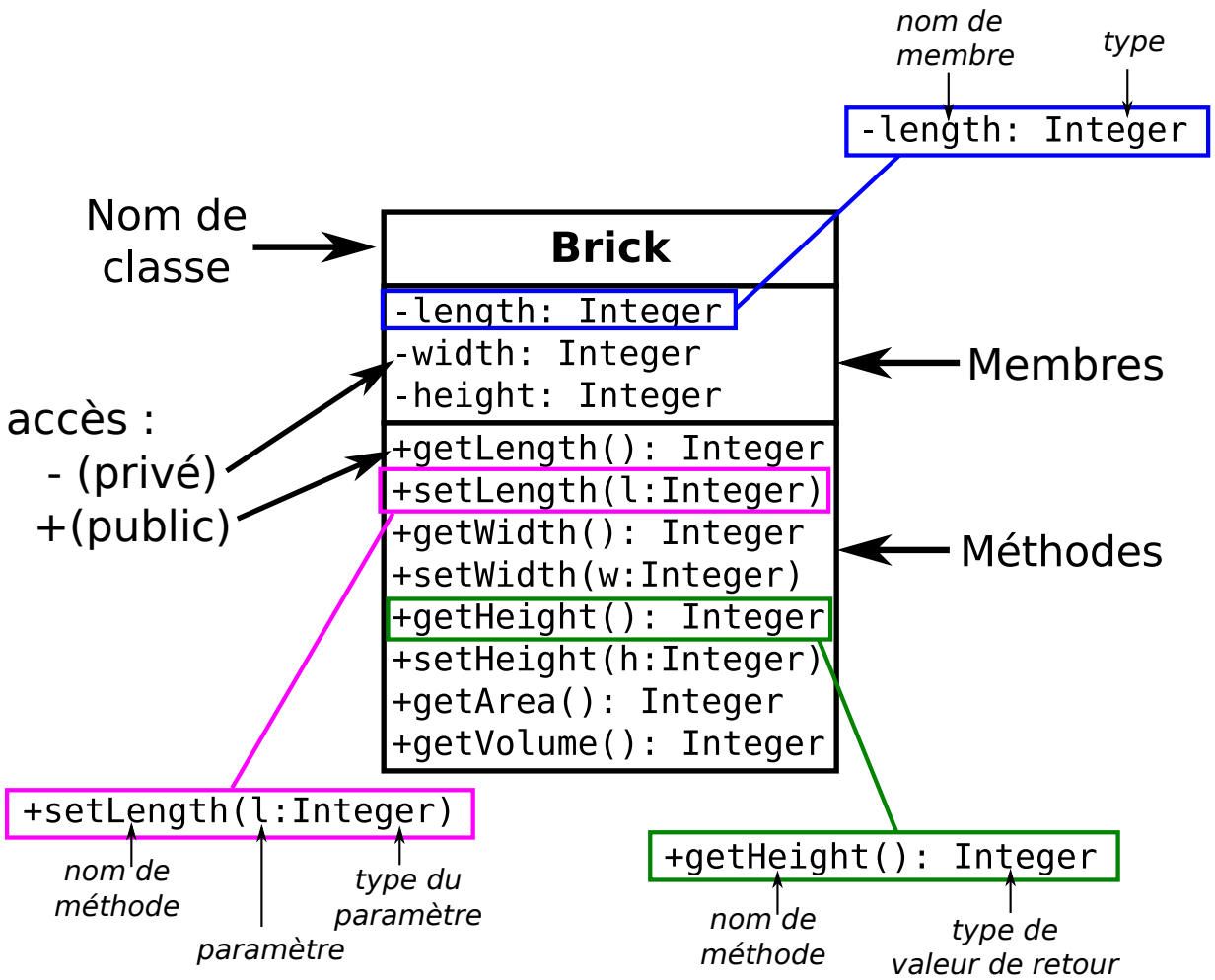
Classe, membres et méthodes : représentation UML



Classe, membres et méthodes : représentation UML



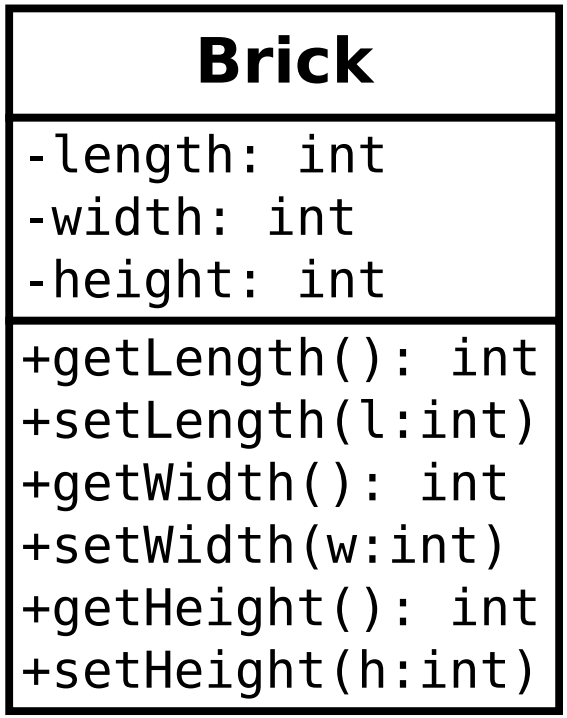
Classe, membres et méthodes : représentation UML



Nom de la méthode, type de retour, nombre et types des arguments = **signature** de la méthode

Implémentation en Java

```java
public class Brick
{
    private int length;
    private int width;
    private int height;

    public int getLength()
    {
        return length;
    }

    public void setLength(int l)
    {
        length = l;
    }

    public int getWidth()
    {
        return width;
    }

    public void setWidth(int w)
    {
        width = w;
    }

    public int getHeight()
    {
        return height;
    }

    public void setHeight(int h)
    {
        height = h;
    }
}
```



Méthodes : conception et utilisation

Concepteur de la classe

Brick
-length: Integer -width: Integer -height: Integer
+getLength(): Integer +setLength(l:Integer) +getWidth(): Integer +setWidth(w:Integer) +getHeight(): Integer +setHeight(h:Integer)

Utilisateur de la classe

```
Brick blueBrick = new Brick();

blueBrick.setLength(2);
blueBrick.setWidth(3);
blueBrick.setHeight(1);

int area = blueBrick.getLength() * blueBrick.getWidth();
int volume  = area * blueBrick.getHeight();
```

Méthodes : conception et utilisation

Concepteur de la classe

Brick
-length: Integer -width: Integer -height: Integer
+getLength(): Integer +setLength(l:Integer) +getWidth(): Integer +setWidth(w:Integer) +getHeight(): Integer +setHeight(h:Integer)

Utilisateur de la classe

```
Brick blueBrick = new Brick();

blueBrick.setLength(2);
blueBrick.setWidth(3);
blueBrick.setHeight(1);

int area = blueBrick.getArea();
int volume  = blueBrick.getVolume();
```

Méthodes : conception et utilisation

Concepteur de la classe

Brick
-length: Integer -width: Integer -height: Integer
+getLength(): Integer +setLength(l:Integer) +getWidth(): Integer +setWidth(w:Integer) +getHeight(): Integer +setHeight(h:Integer) +getArea(): Integer +getVolume(): Integer

Utilisateur de la classe

```
Brick blueBrick = new Brick();

blueBrick.setLength(2);
blueBrick.setWidth(3);
blueBrick.setHeight(1);

int area = blueBrick.getArea();
int volume  = blueBrick.getVolume();
```

Méthodes : conception et utilisation

Concepteur de la classe

Brick
-length: Integer -width: Integer -height: Integer
+getLength(): Integer +setLength(l:Integer) +getWidth(): Integer +setWidth(w:Integer) +getHeight(): Integer +setHeight(h:Integer) +getArea(): Integer +getVolume(): Integer

```
public int getArea ()
{
    return length * width;
}

public int getVolume ()
{
    return length * width * height;
}
```

Utilisateur de la classe

```
Brick blueBrick = new Brick();

blueBrick.setLength(2);
blueBrick.setWidth(3);
blueBrick.setHeight(1);

int area = blueBrick.getArea();
int volume  = blueBrick.getVolume();
```

Méthodes : conception et utilisation

Concepteur de la classe

Brick
-length: Integer -width: Integer -height: Integer
+getLength(): Integer +setLength(l:Integer) +getWidth(): Integer +setWidth(w:Integer) +getHeight(): Integer +setHeight(h:Integer) +getArea(): Integer +getVolume(): Integer

```
public int getArea ()
{
    return length * width;
}
```

~~public int getVolume ()~~
~~{~~
~~return length * width * height;~~
~~}~~

Utilisateur de la classe

```
Brick blueBrick = new Brick();

blueBrick.setLength(2);
blueBrick.setWidth(3);
blueBrick.setHeight(1);

int area = blueBrick.getArea();
int volume  = blueBrick.getVolume();
```

Changement d'implémentation

```
public int getVolume ()
{
    return getArea() * height;
}
```

Méthodes : conception et utilisation

Concepteur de la classe

Brick
-length: Integer -width: Integer -height: Integer
+getLength(): Integer +setLength(l:Integer) +getWidth(): Integer +setWidth(w:Integer) +getHeight(): Integer +setHeight(h:Integer) +getArea(): Integer +getVolume(): Integer

```
public int getArea ()
{
    return length * width;
}
```

~~public int getVolume ()~~
~~{~~
~~return length * width * height;~~
~~}~~

Utilisateur de la classe

```
Brick blueBrick = new Brick();

blueBrick.setLength(2);
blueBrick.setWidth(3);
blueBrick.setHeight(1);

int area = blueBrick.getArea();
int volume  = blueBrick.getVolume();
```

reste valide

Changement d'implémentation

```
public int getVolume ()
{
    return getArea() * height;
}
```

Méthodes : conception et utilisation

Concepteur de la classe

Brick
-length: Integer -width: Integer -height: Integer
+getLength(): Integer +setLength(l:Integer) +getWidth(): Integer +setWidth(w:Integer) +getHeight(): Integer +setHeight(h:Integer) +getArea(): Integer +getVolume(): Integer

```
public int getArea ()
{
    return length * width;
}
```

```
public int getVolume ()
{
    return getArea() * height;
}
```

Utilisateur de la classe

```
Brick blueBrick = new Brick();

blueBrick.setLength(2);
blueBrick.setWidth(3);
blueBrick.setHeight(1);

int area = blueBrick.getArea();
int volume  = blueBrick.getVolume();
```

utilisation de l'objet à l'extérieur de la classe : invocation de la méthode sur l'objet

`nomObjet.nomMethode()`

Méthodes : conception et utilisation

Concepteur de la classe

Brick
-length: Integer -width: Integer -height: Integer
+getLength(): Integer +setLength(l:Integer) +getWidth(): Integer +setWidth(w:Integer) +getHeight(): Integer +setHeight(h:Integer) +getArea(): Integer +getVolume(): Integer

```
public int getArea ()
{
    return length * width;
}
```

```
public int getVolume ()
{
    return getArea() * height;
}
```

à l'intérieur de la classe : invocation de la méthode d'un objet ***sur lui-même***

`nomMethode()`

Utilisateur de la classe

```
Brick blueBrick = new Brick();

blueBrick.setLength(2);
blueBrick.setWidth(3);
blueBrick.setHeight(1);

int area = blueBrick.getArea();
int volume  = blueBrick.getVolume();
```

utilisation de l'objet à l'extérieur de la classe : invocation de la méthode sur l'objet

`nomObjet.nomMethode()`

Méthodes = interface d'un objet

Concepteur de la classe

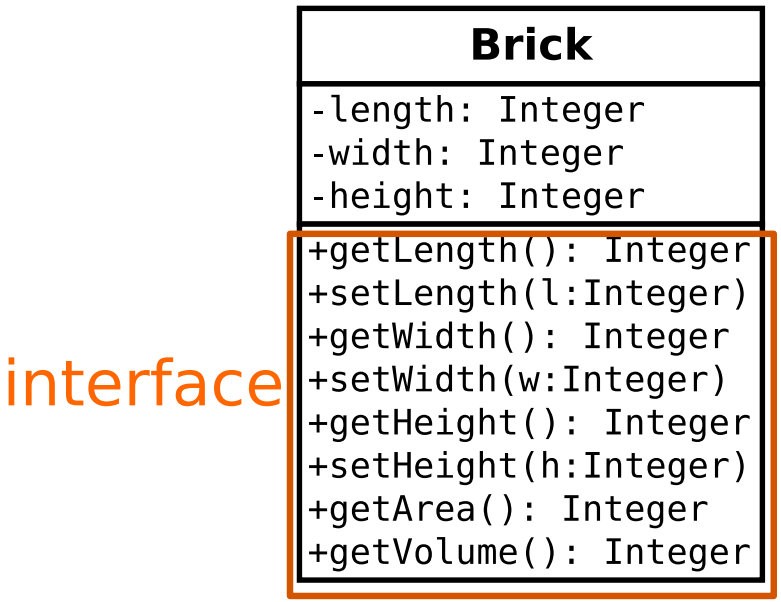
Brick
-length: Integer -width: Integer -height: Integer
+getLength(): Integer +setLength(l:Integer) +getWidth(): Integer +setWidth(w:Integer) +getHeight(): Integer +setHeight(h:Integer) +getArea(): Integer +getVolume(): Integer

Utilisateur de la classe

```
Brick blueBrick = new Brick();

blueBrick.setLength(2);
blueBrick.setWidth(3);
blueBrick.setHeight(1);


int volume  = blueBrick.getVolume();
```

Méthodes = interface d'un objet

Concepteur de la classe



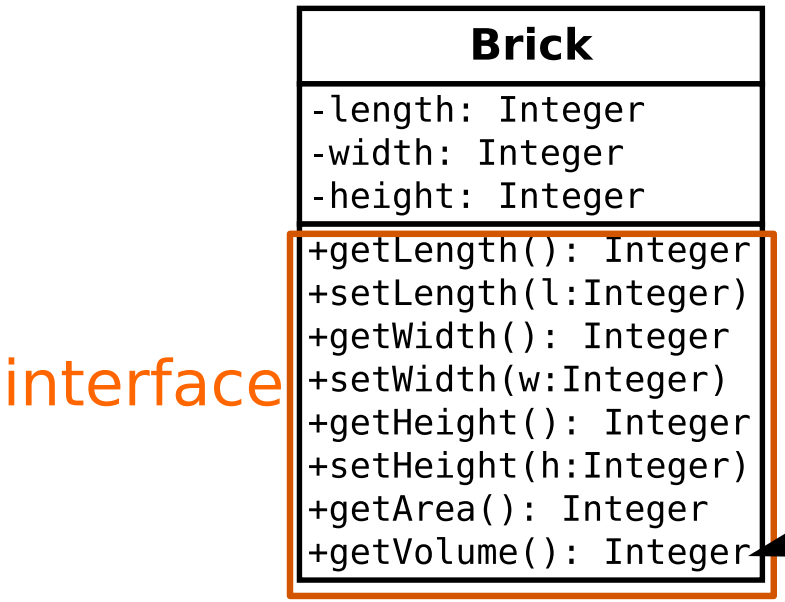
Utilisateur de la classe

```
Brick blueBrick = new Brick();

blueBrick.setLength(2);
blueBrick.setWidth(3);
blueBrick.setHeight(1);


int volume  = blueBrick.getVolume();
```

Méthodes = interface d'un objet

Concepteur de la classe

Utilisateur de la classe

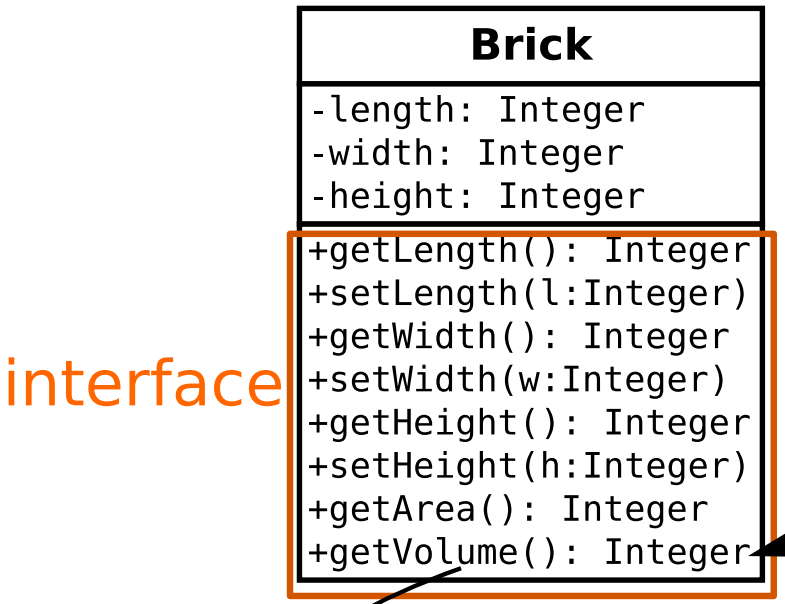



```
Brick blueBrick = new Brick();

blueBrick.setLength(2);
blueBrick.setWidth(3);
blueBrick.setHeight(1);


int volume  = blueBrick.getVolume();
```

Méthodes = interface d'un objet

Concepteur de la classe

Utilisateur de la classe



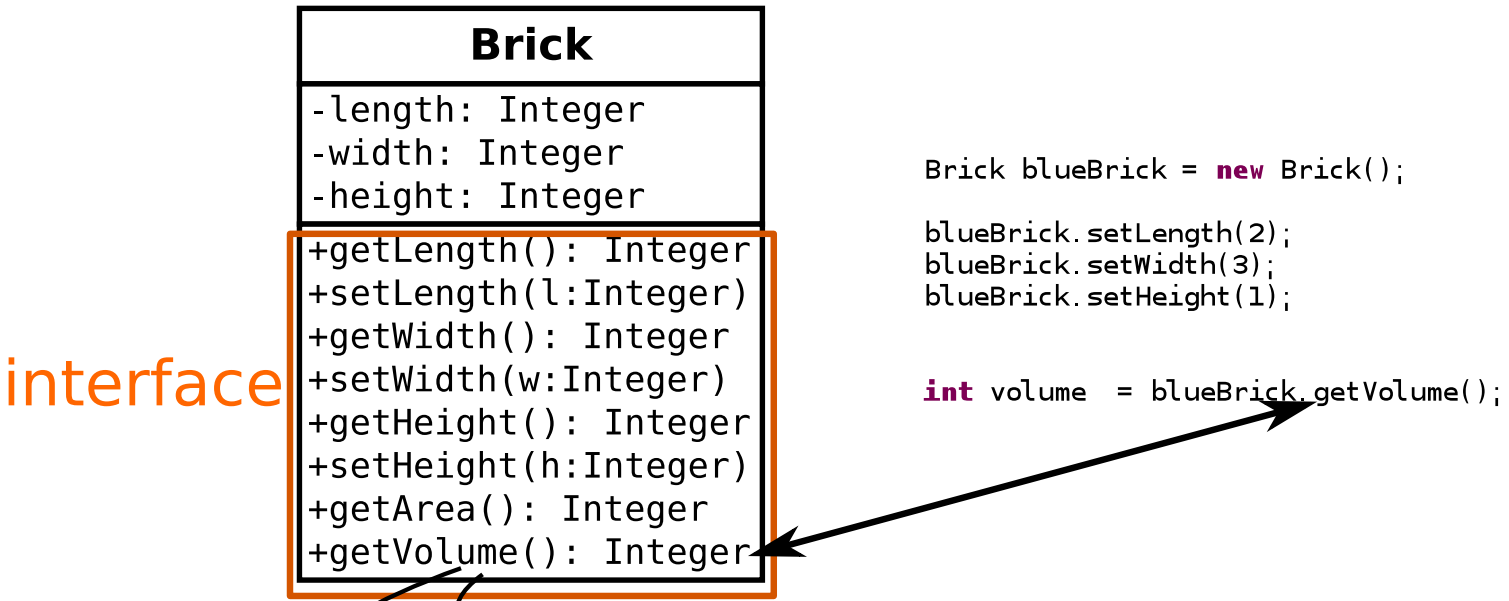

```
Brick blueBrick = new Brick();

blueBrick.setLength(2);
blueBrick.setWidth(3);
blueBrick.setHeight(1);


int volume  = blueBrick.getVolume();
```

```
public int getVolume ()
{
    return length * width * height;
}
```

Méthodes = interface d'un objet

Concepteur de la classe

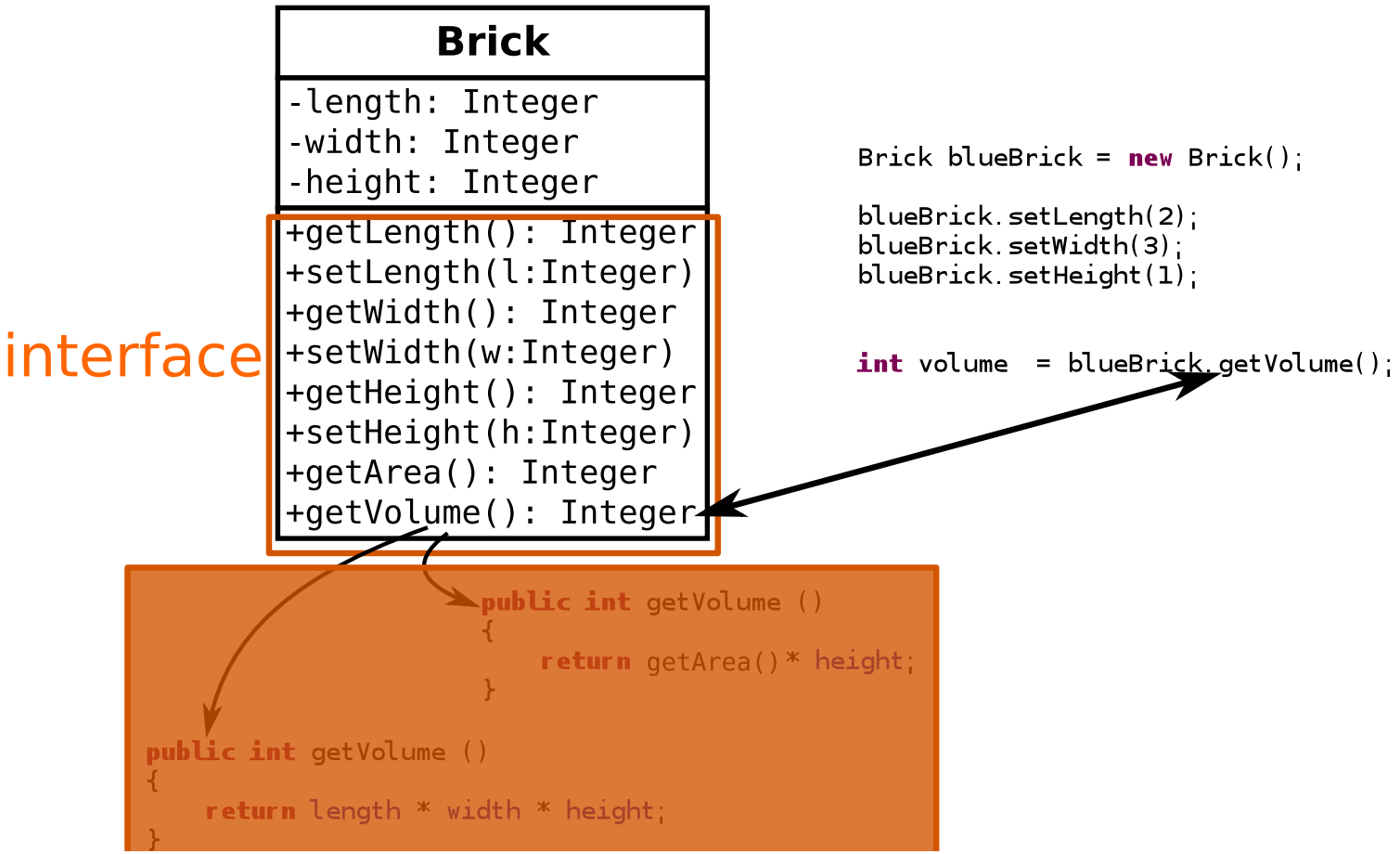
Utilisateur de la classe



Méthodes = interface d'un objet

Concepteur de la classe

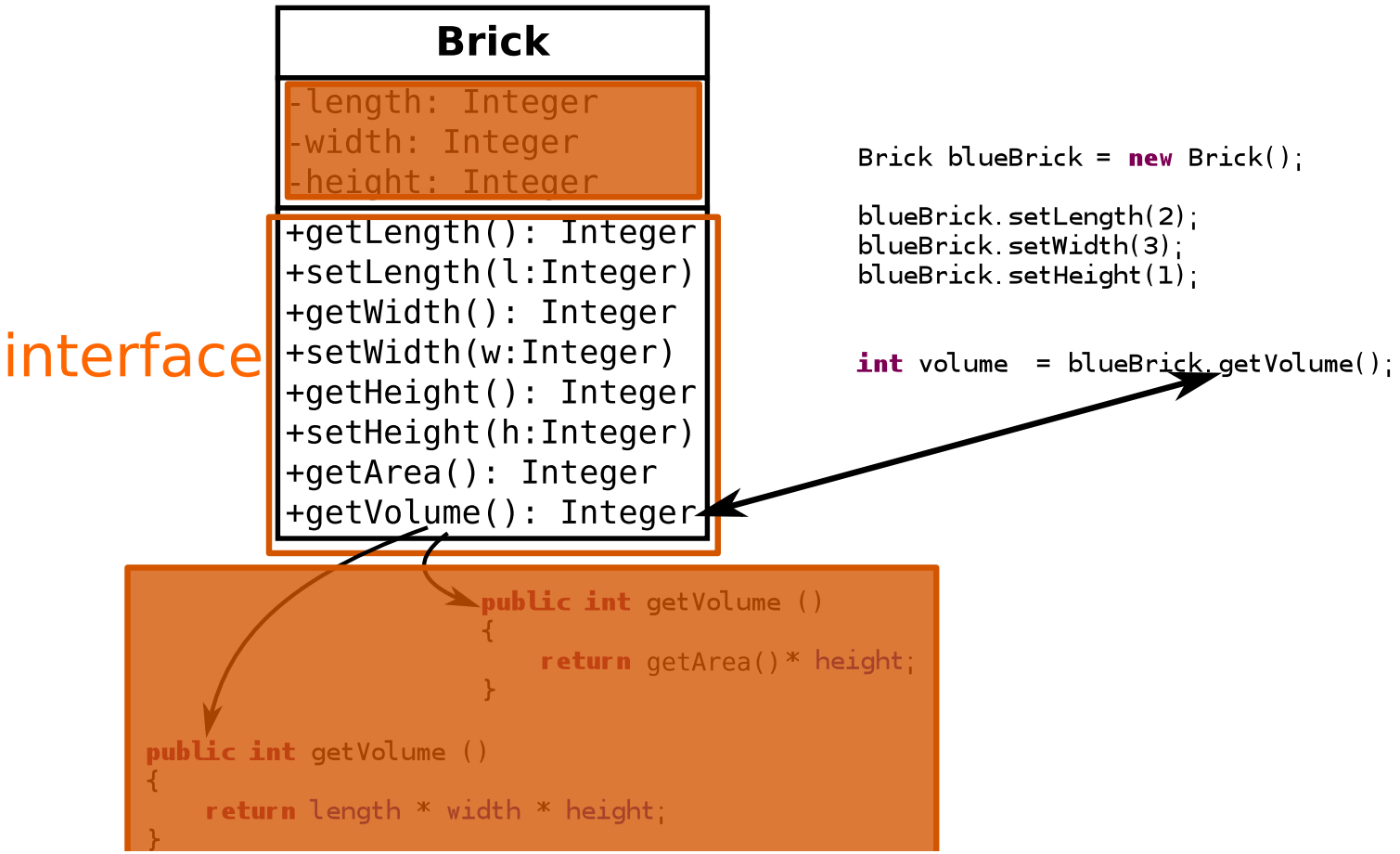
Utilisateur de la classe



Méthodes = interface d'un objet

Concepteur de la classe

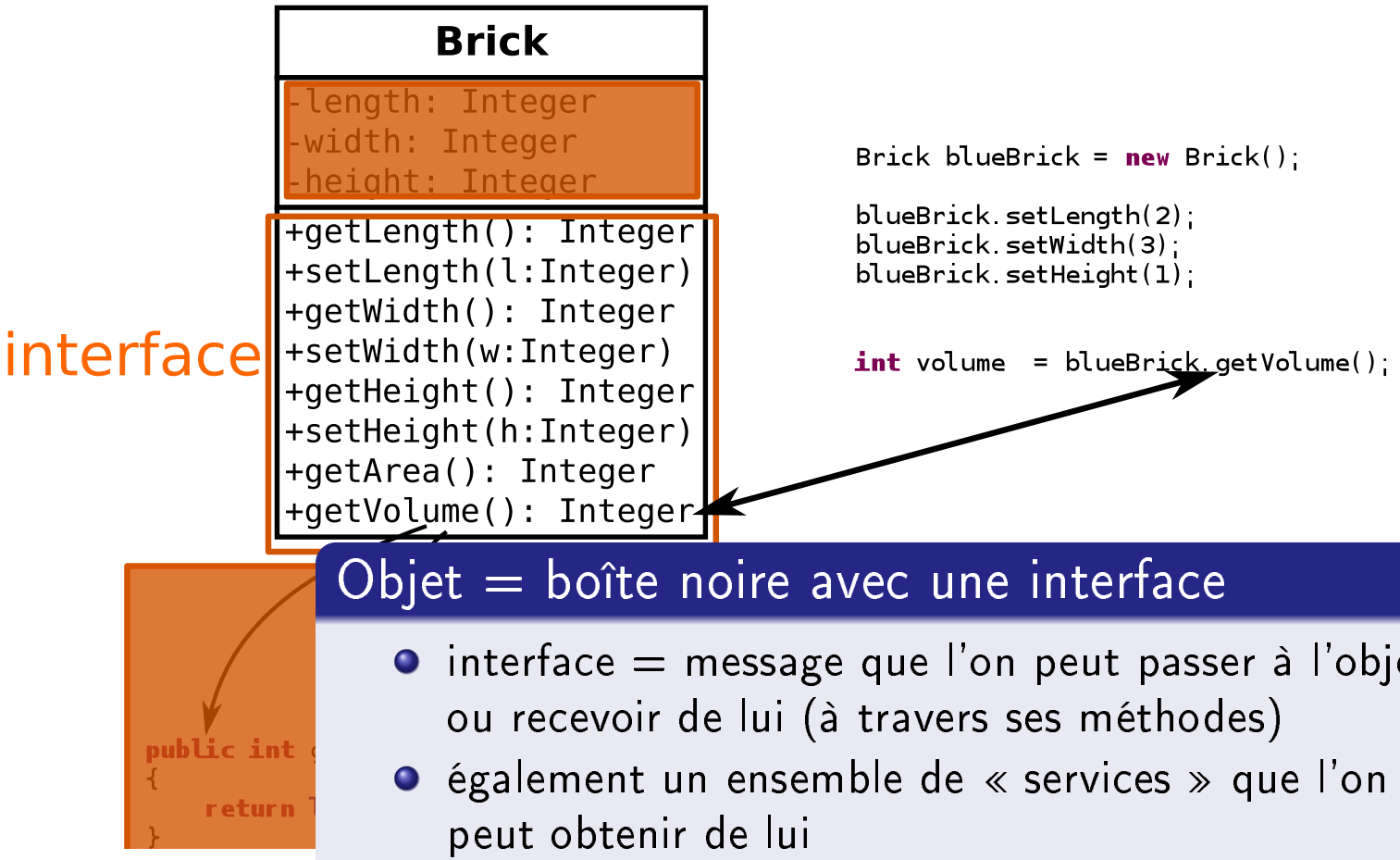
Utilisateur de la classe



Méthodes = interface d'un objet

Concepteur de la classe

Utilisateur de la classe



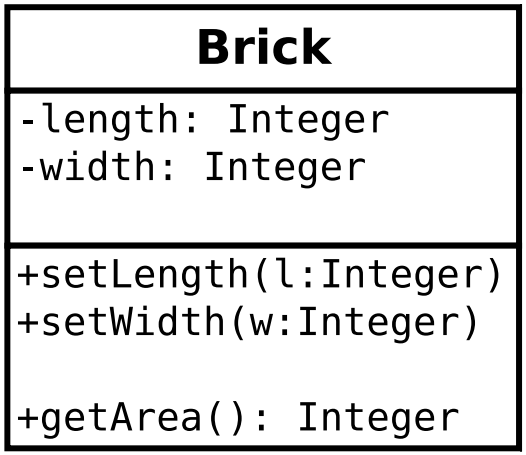
Objet = boîte noire avec une interface

- interface = message que l'on peut passer à l'objet, ou recevoir de lui (à travers ses méthodes)
- également un ensemble de « services » que l'on peut obtenir de lui
- implémentation = comment réaliser ces services

Encapsulation : cohérence d'un objet

```java
public class Brick
{
    private int length;
    private int width;

    public void setLength(int l)
    {
        length = l;
    }
    public void setWidth(int w)
    {
        width = w;
    }
    public int getArea ()
    {
        return length * width;
    }
}
```

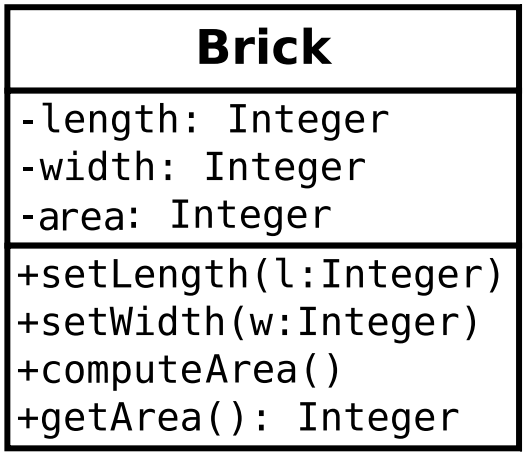
```java
Brick blueBrick = new Brick();

blueBrick.setLength(2);
blueBrick.setWidth(3);

int area = blueBrick.getArea();
System.out.println("Aire = " + String.valueOf(area));
```



Encapsulation : cohérence d'un objet

```java
public class Brick
{
    private int length;
    private int width;
    private int area;

    public void setLength(int l)
    {
        length = l;
    }

    public void setWidth(int w)
    {
        width = w;
    }

    public void computeArea ()
    {
        area = length * width;
    }

    public int getArea ()
    {
        return area;
    }
}
```

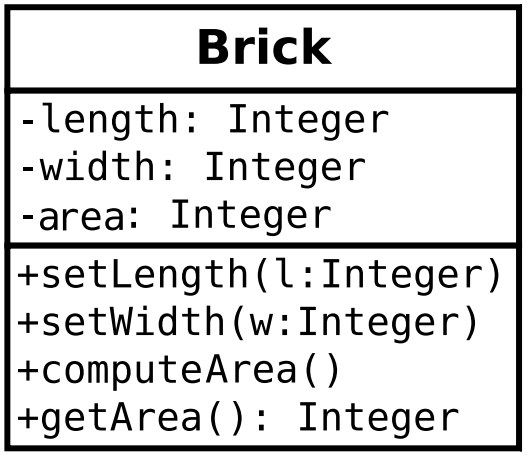
```java
Brick blueBrick = new Brick();

blueBrick.setLength(2);
blueBrick.setWidth(3);
blueBrick.computeArea();

int area = blueBrick.getArea();
System.out.println("Aire = " + String.valueOf(area));
```



Encapsulation : cohérence d'un objet

```java
public class Brick
{
    private int length;
    private int width;
    private int area;

    public void setLength(int l)
    {
        length = l;
    }

    public void setWidth(int w)
    {
        width = w;
    }

    public void computeArea ()
    {
        area = length * width;
    }

    public int getArea ()
    {
        return area;
    }
}
```

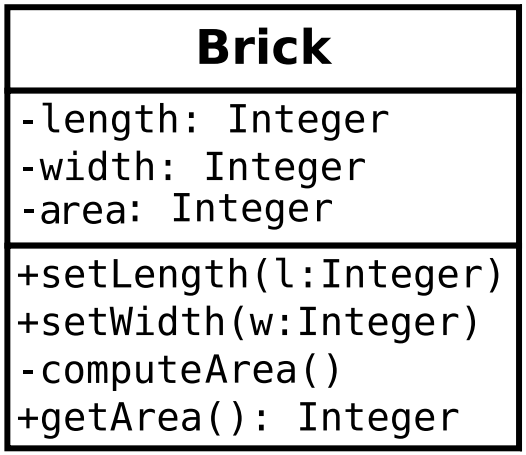
```java
Brick blueBrick = new Brick();

blueBrick.setLength(2);
blueBrick.setWidth(3);
blueBrick.computeArea();

int area = blueBrick.getArea();
System.out.println("Aire = " + String.valueOf(area));

blueBrick.setLength(3);
area = blueBrick.getArea();
System.out.println("Aire = " + String.valueOf(area));
```



Encapsulation : cohérence d'un objet

```java
public class Brick
{
    private int length;
    private int width;
    private int area;

    public void setLength(int l)
    {
        length = l;
        computeArea();
    }

    public void setWidth(int w)
    {
        width = w;
        computeArea();
    }

    private void computeArea ()
    {
        area = length * width;
    }

    public int getArea ()
    {
        return area;
    }
}
```

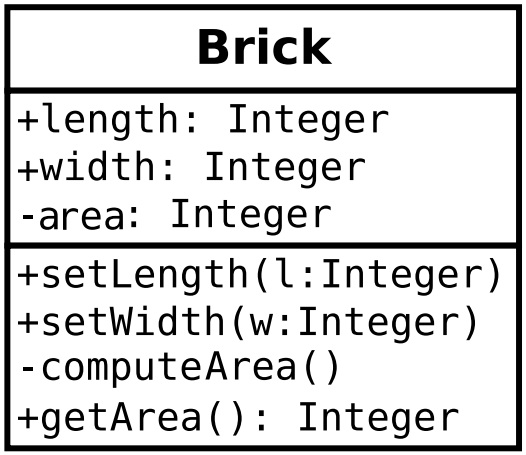
```java
Brick blueBrick = new Brick();

blueBrick.setLength(2);
blueBrick.setWidth(3);

int area = blueBrick.getArea();
System.out.println("Aire = " + String.valueOf(area));

blueBrick.setLength(3);
area = blueBrick.getArea();
System.out.println("Aire = " + String.valueOf(area));
```



Encapsulation : cohérence d'un objet

```java
public class Brick
{
    public int length;
    public int width;
    private int area;

    public void setLength(int l)
    {
        length = l;
        computeArea();
    }

    public void setWidth(int w)
    {
        width = w;
        computeArea();
    }

    private void computeArea ()
    {
        area = length * width;
    }

    public int getArea ()
    {
        return area;
    }
}
```

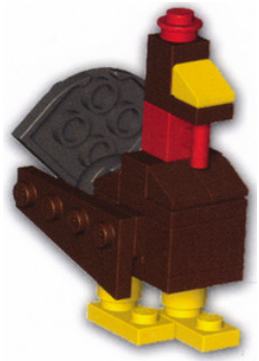
```java
Brick blueBrick = new Brick();

blueBrick.setLength(2);
blueBrick.setWidth(3);

int area = blueBrick.getArea();
System.out.println("Aire = " + String.valueOf(area));

blueBrick.width = 5;
area = blueBrick.getArea();
System.out.println("Aire = " + String.valueOf(area));
```

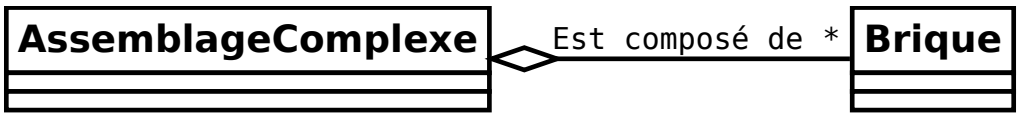


Légo

Modélisation d'entités et de leur comportement :

- décomposition d'un problème en sous-problèmes $\leftrightarrow$ faire des objets aussi petits que possibles, qui représentent des entités/notions (objets du monde réel ou concepts) élémentaires
- ces briques élémentaires peuvent être assemblées pour créer des objets plus complexes





Des objets pour factoriser le code

Et la couleur dans tout ça ? (un couleur = 3 valeurs R, V, B entre 0 et 255)

Brick
-red: Integer -green: Integer -blue: Integer
+setRed(r:Integer) +setGreen(g:Integer) +setBlue(b:Integer) +getRed(): Integer +getGreen(): Integer +getBlue(): Integer

Des objets pour factoriser le code

Et la couleur dans tout ça ? (un couleur = 3 valeurs R, V, B entre 0 et 255)

Brick
-red: Integer -green: Integer -blue: Integer
+setRed(r:Integer) +setGreen(g:Integer) +setBlue(b:Integer) +getRed(): Integer +getGreen(): Integer +getBlue(): Integer

LegoFigurine
-redFace: Integer -greenFace: Integer -blueFace: Integer
+setRedFace(r:Integer) +setGreenFace(v:Integer) +setBlueFace(v:Integer)

Des objets pour factoriser le code

Et la couleur dans tout ça ? (un couleur = 3 valeurs R, V, B entre 0 et 255)

Brick
-red: Integer -green: Integer -blue: Integer
+setRed(r:Integer) +setGreen(g:Integer) +setBlue(b:Integer) +getRed(): Integer +getGreen(): Integer +getBlue(): Integer

LegoFigurine
-redFace: Integer -greenFace: Integer -blueFace: Integer
+setRedFace(r:Integer) +setGreenFace(v:Integer) +setBlueFace(v:Integer)

Des objets pour factoriser le code

Et la couleur dans tout ça ? (un couleur = 3 valeurs R, V, B entre 0 et 255)

Brick
-red: Integer -green: Integer -blue: Integer
+setRed(r:Integer) +setGreen(g:Integer) +setBlue(b:Integer) +getRed(): Integer +getGreen(): Integer +getBlue(): Integer +highlight()

LegoFigurine
-redFace: Integer -greenFace: Integer -blueFace: Integer
+setRedFace(r:Integer) +setGreenFace(v:Integer) +setBlueFace(v:Integer)

Des objets pour factoriser le code

Et la couleur dans tout ça ? (un couleur = 3 valeurs R, V, B entre 0 et 255)

Brick

```
-red: Integer
-green: Integer
-blue: Integer
```

```
+setRed(r:Integer)
+setGreen(g:Integer)
+setBlue(b:Integer)
+getRed(): Integer
+getGreen(): Integer
+getBlue(): Integer
+highlight()
```

```
highlight()
red = 255 - red;
green = 255 - green;
blue = 255 - blue;
```

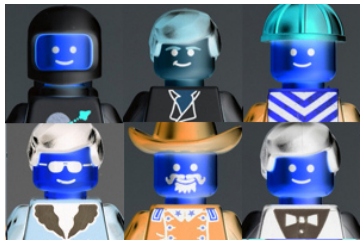
LegoFigurine

```
-redFace: Integer
-greenFace: Integer
-blueFace: Integer
```

```
+setRedFace(r:Integer)
+setGreenFace(v:Integer)
+setBlueFace(v:Integer)
```

Des objets pour factoriser le code

Et la couleur dans tout ça ? (un couleur = 3 valeurs R, V, B entre 0 et 255)



Brick

```
-red: Integer
-green: Integer
-blue: Integer
```

```
+setRed(r:Integer)
+setGreen(g:Integer)
+setBlue(b:Integer)
+getRed(): Integer
+getGreen(): Integer
+getBlue(): Integer
+highlight()
```

```
highlight()
red = 255 - red;
green = 255 - green;
blue = 255 - blue;
```

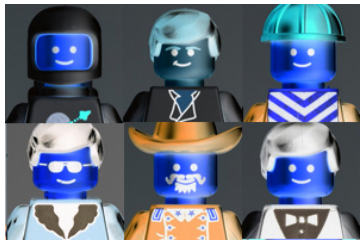
LegoFigurine

```
-redFace: Integer
-greenFace: Integer
-blueFace: Integer
```

```
+setRedFace(r:Integer)
+setGreenFace(v:Integer)
+setBlueFace(v:Integer)
```

Des objets pour factoriser le code

Et la couleur dans tout ça ? (un couleur = 3 valeurs R, V, B entre 0 et 255)



Brick

```
-red: Integer
-green: Integer
-blue: Integer
+setRed(r:Integer)
+setGreen(g:Integer)
+setBlue(b:Integer)
+getRed(): Integer
+getGreen(): Integer
+getBlue(): Integer
+highlight()
```

```
highlight()
red = 255 - red;
green = 255 - green;
blue = 255 - blue;
```

LegoFigurine

```
-redFace: Integer
-greenFace: Integer
-blueFace: Integer
+setRedFace(r:Integer)
+setGreenFace(v:Integer)
+setBlueFace(v:Integer)
+highlight()
```

Des objets pour factoriser le code

Et la couleur dans tout ça ? (un couleur = 3 valeurs R, V, B entre 0 et 255)

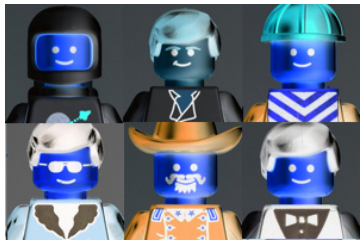
Brick

-red: Integer
-green: Integer
-blue: Integer

+setRed(r:Integer)
+setGreen(g:Integer)
+setBlue(b:Integer)
+getRed(): Integer
+getGreen(): Integer
+getBlue(): Integer
+highlight()

highlight()

```
red = 255 - red;
green = 255 - green;
blue = 255 - blue;
```



LegoFigurine

-redFace: Integer
-greenFace: Integer
-blueFace: Integer

+setRedFace(r:Integer)
+setGreenFace(v:Integer)
+setBlueFace(v:Integer)

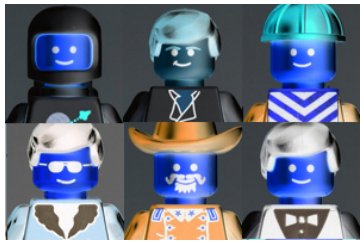
+highlight()

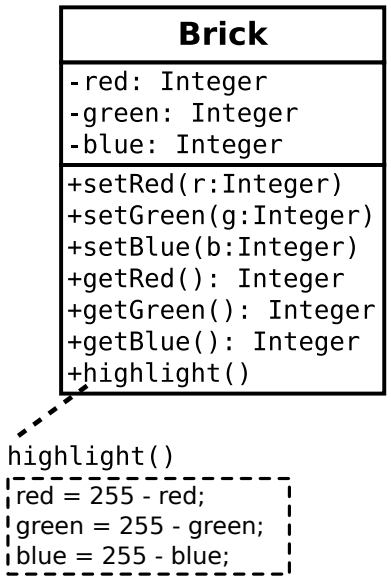
highlight()

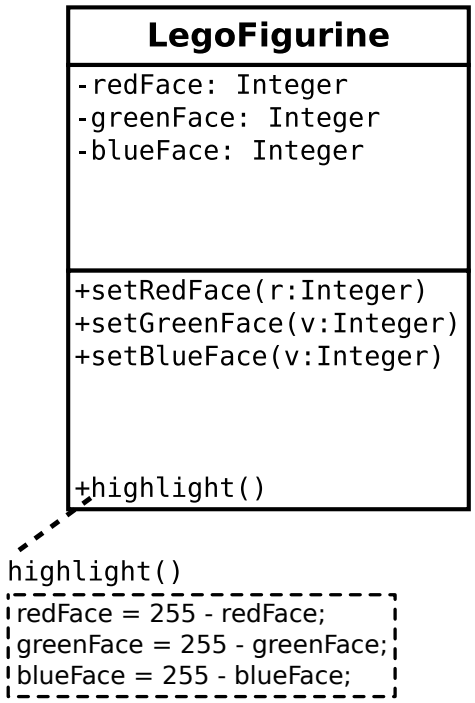
```
redFace = 255 - redFace;
greenFace = 255 - greenFace;
blueFace = 255 - blueFace;
```

Des objets pour factoriser le code

Et la couleur dans tout ça ? (un couleur = 3 valeurs R, V, B entre 0 et 255)



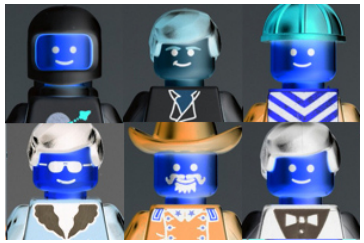


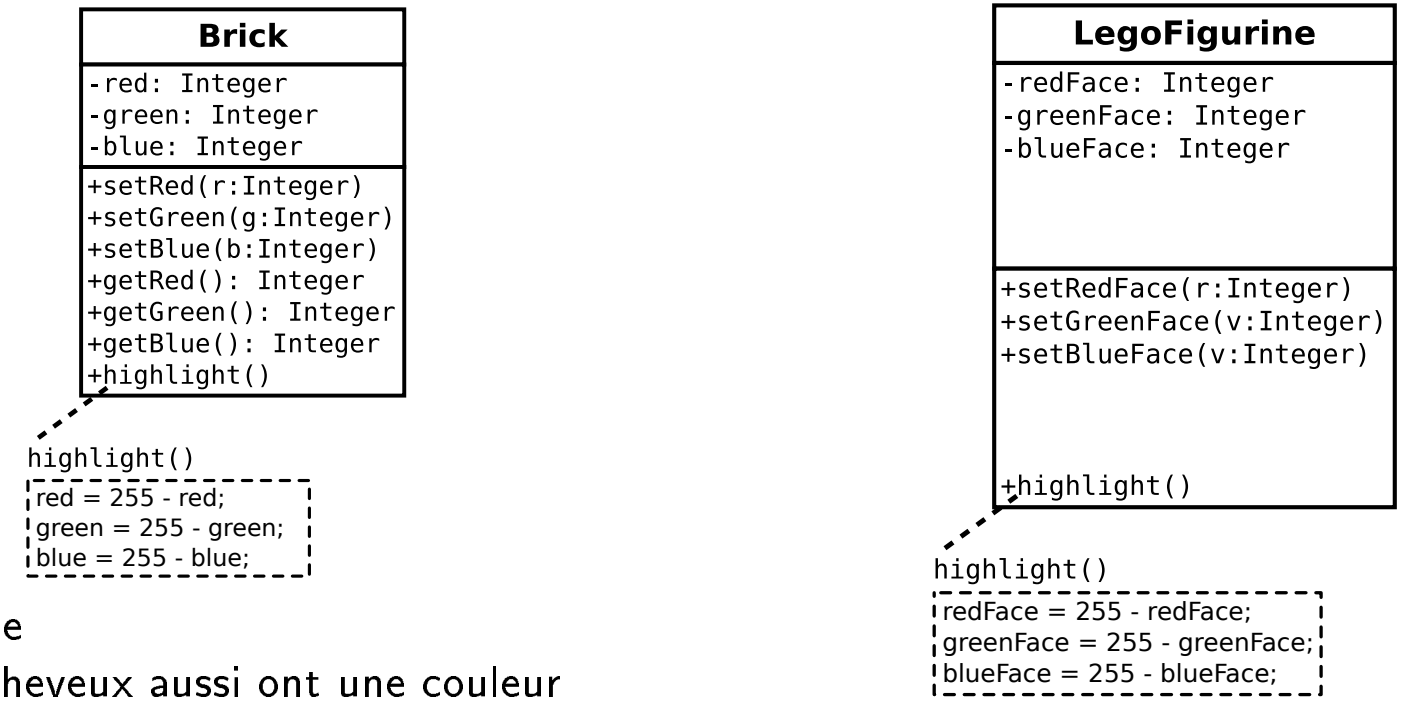


- redite

Des objets pour factoriser le code

Et la couleur dans tout ça ? (un couleur = 3 valeurs R, V, B entre 0 et 255)

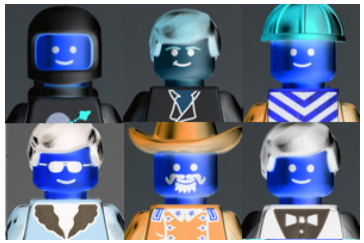




- redite
- les cheveux aussi ont une couleur

Des objets pour factoriser le code

Et la couleur dans tout ça ? (un couleur = 3 valeurs R, V, B entre 0 et 255)



Brick

```
-red: Integer
-green: Integer
-blue: Integer
+setRed(r:Integer)
+setGreen(g:Integer)
+setBlue(b:Integer)
+getRed(): Integer
+getGreen(): Integer
+getBlue(): Integer
+highlight()
```

```
highlight()
red = 255 - red;
green = 255 - green;
blue = 255 - blue;
```

LegoFigurine

```
-redFace: Integer
-greenFace: Integer
-blueFace: Integer
-redHair: Integer
-greenHair: Integer
-blueHair: Integer
+setRedFace(r:Integer)
+setGreenFace(v:Integer)
+setBlueFace(v:Integer)
+setRedHair(r:Integer)
+setGreenHair(g:Integer)
+setBlueHair(r:Integer)
+highlight()
```

```
highlight()
redFace = 255 - redFace;
greenFace = 255 - greenFace;
blueFace = 255 - blueFace;
redHair = 255 - redHair;
greenHair = 255 - greenHair;
blueHair = 255 - blueHair;
```

- redite
- les cheveux aussi ont une couleur

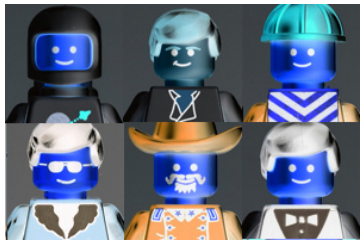
Des objets pour factoriser le code

Et la couleur dans tout ça ? (un couleur = 3 valeurs R, V, B entre 0 et 255)

Brick

```
-red: Integer
-green: Integer
-blue: Integer
+setRed(r:Integer)
+setGreen(g:Integer)
+setBlue(b:Integer)
+getRed(): Integer
+getGreen(): Integer
+getBlue(): Integer
+highlight()
```

```
highlight()
red = 255 - red;
green = 255 - green;
blue = 255 - blue;
```



LegoFigurine

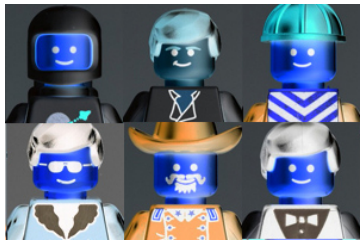
```
-redFace: Integer
-greenFace: Integer
-blueFace: Integer
-redHair: Integer
-greenHair: Integer
-blueHair: Integer
+setRedFace(r:Integer)
+setGreenFace(v:Integer)
+setBlueFace(v:Integer)
+setRedHair(r:Integer)
+setGreenHair(g:Integer)
+setBlueHair(r:Integer)
+highlight()
```

```
highlight()
redFace = 255 - redFace;
greenFace = 255 - greenFace;
blueFace = 255 - blueFace;
redHair = 255 - redHair;
greenHair = 255 - greenHair;
blueHair = 255 - blueHair;
```

- redite
- les cheveux aussi ont une couleur
- si on augmentait le nb de nuances ?

Des objets pour factoriser le code

Et la couleur dans tout ça ? (un couleur = 3 valeurs R, V, B entre 0 et 255)



Brick

```
-red: Integer
-green: Integer
-blue: Integer
+setRed(r:Integer)
+setGreen(g:Integer)
+setBlue(b:Integer)
+getRed(): Integer
+getGreen(): Integer
+getBlue(): Integer
+highlight()
```

```
highlight()
red = 255 - red;
green = 1023 - green;
blue = 1023 - blue;
```

LegoFigurine

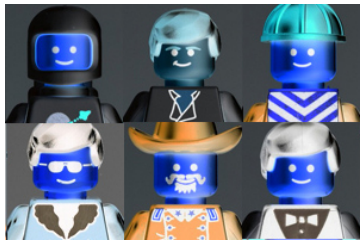
```
-redFace: Integer
-greenFace: Integer
-blueFace: Integer
-redHair: Integer
-greenHair: Integer
-blueHair: Integer
+setRedFace(r:Integer)
+setGreenFace(v:Integer)
+setBlueFace(v:Integer)
+setRedHair(r:Integer)
+setGreenHair(g:Integer)
+setBlueHair(r:Integer)
+highlight()
```

```
highlight()
redFace = 255 - redFace;
redFace = 1023 - redFace;
greenFace = 1023 - greenFace;
blueFace = 1023 - blueFace;
redHair = 1023 - redHair;
greenHair = 1023 - greenHair;
blueHair = 1023 - blueHair;
```

- redite
- les cheveux aussi ont une couleur
- si on augmentait le nb de nuances ?

Des objets pour factoriser le code

Et la couleur dans tout ça ? (un couleur = 3 valeurs R, V, B entre 0 et 255)



Brick

```
-red: Integer
-green: Integer
-blue: Integer
+setRed(r:Integer)
+setGreen(g:Integer)
+setBlue(b:Integer)
+getRed(): Integer
+getGreen(): Integer
+getBlue(): Integer
+highlight()
```

```
highlight()
red = 255 - red;
green = 1023 - green;
blue = 1023 - blue;
```

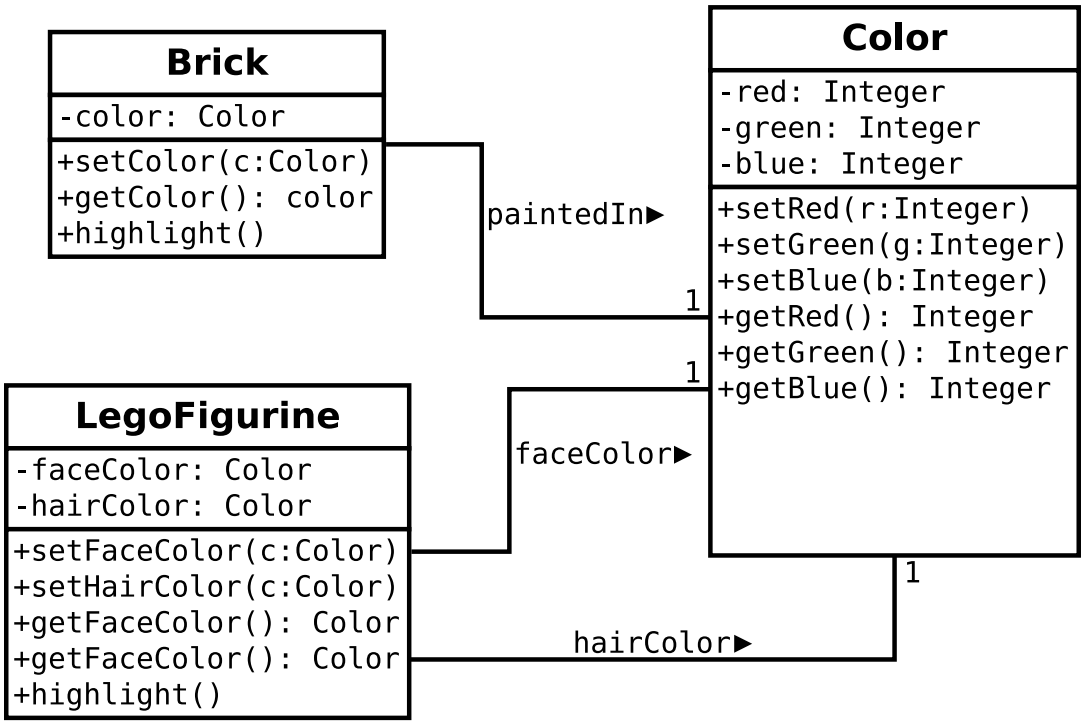
LegoFigurine

```
-redFace: Integer
-greenFace: Integer
-blueFace: Integer
-redHair: Integer
-greenHair: Integer
-blueHair: Integer
+setRedFace(r:Integer)
+setGreenFace(v:Integer)
+setBlueFace(v:Integer)
+setRedHair(r:Integer)
+setGreenHair(g:Integer)
+setBlueHair(r:Integer)
+highlight()
```

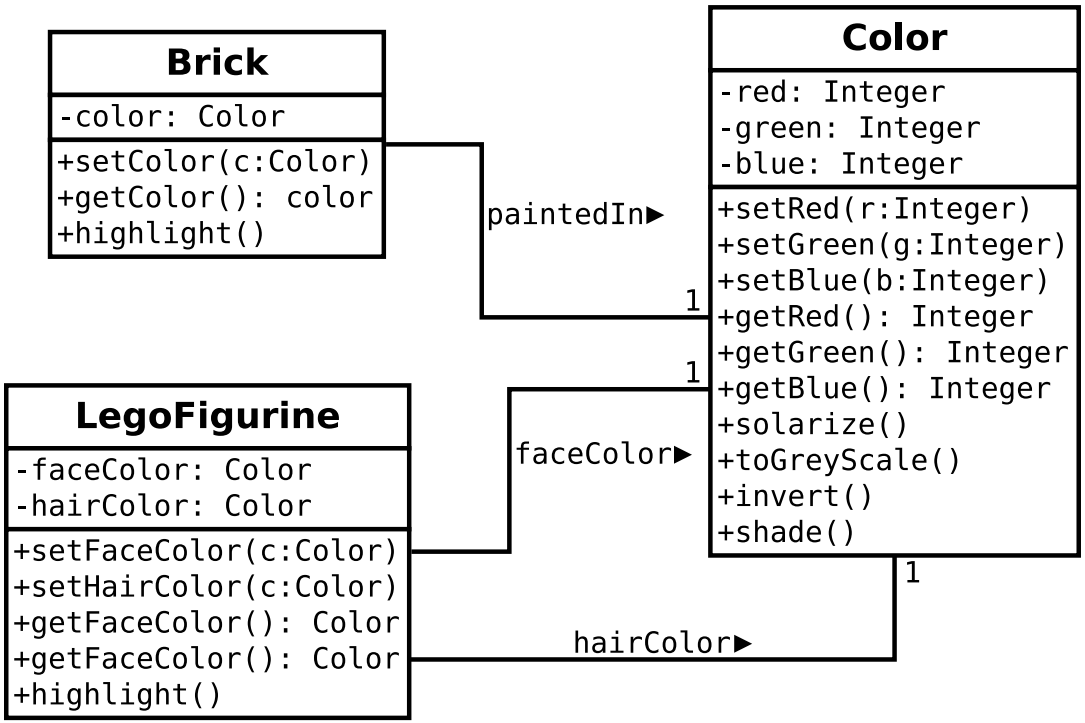
```
highlight()
redFace = 255 - redFace;
redFace = 1023 - redFace;
greenFace = 1023 - greenFace;
blueFace = 1023 - blueFace;
redHair = 1023 - redHair;
greenHair = 1023 - greenHair;
blueHair = 1023 - blueHair;
```

- redite
- les cheveux aussi ont une couleur
- si on augmentait le nb de nuances ?
- si on changeait le rendu de `highlight()` ?

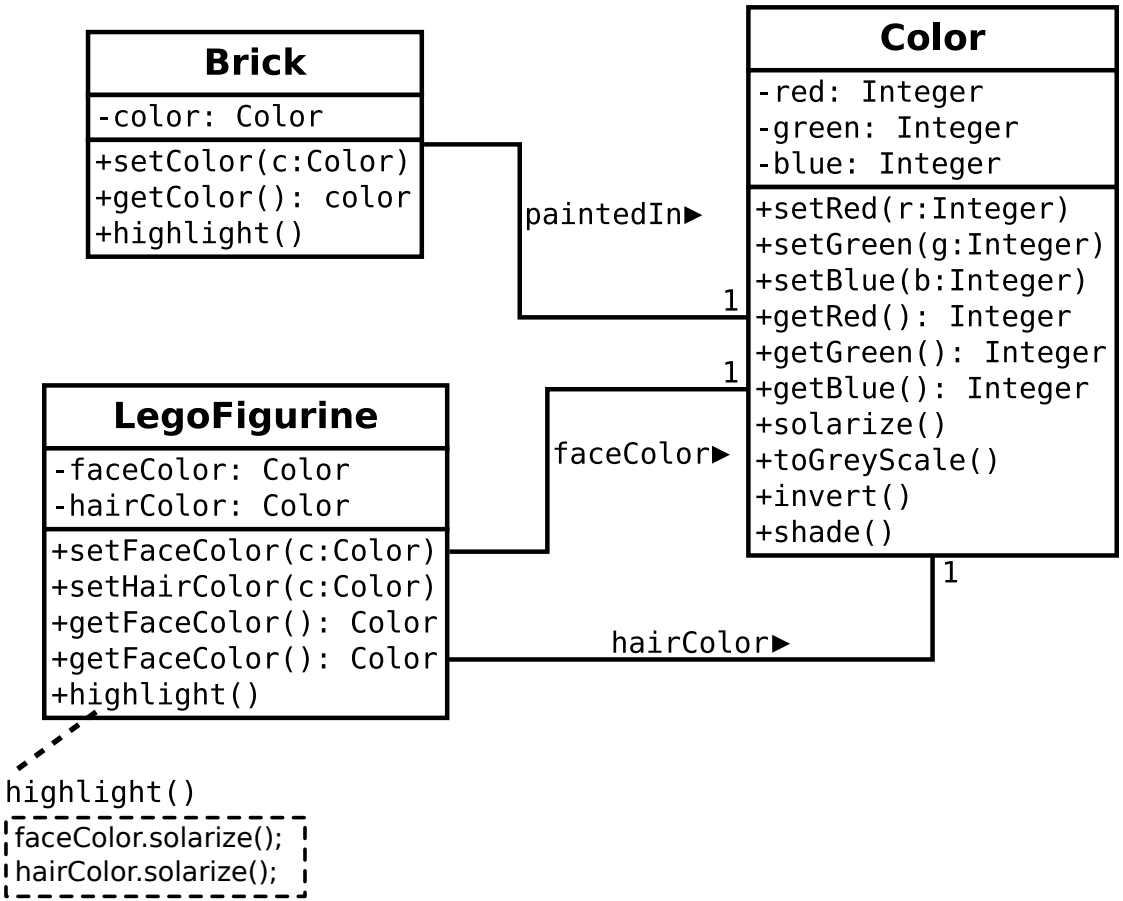
Des objets pour factoriser le code



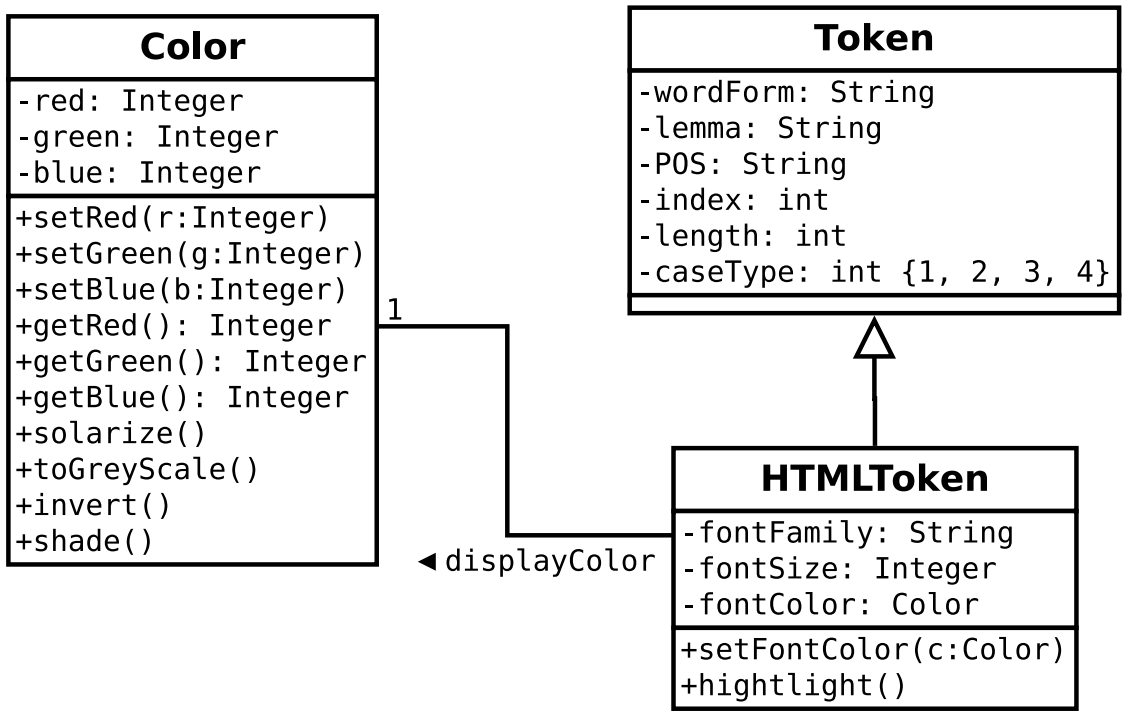
Des objets pour factoriser le code



Des objets pour factoriser le code



Des objets pour factoriser le code



Factoriser... et réutiliser !