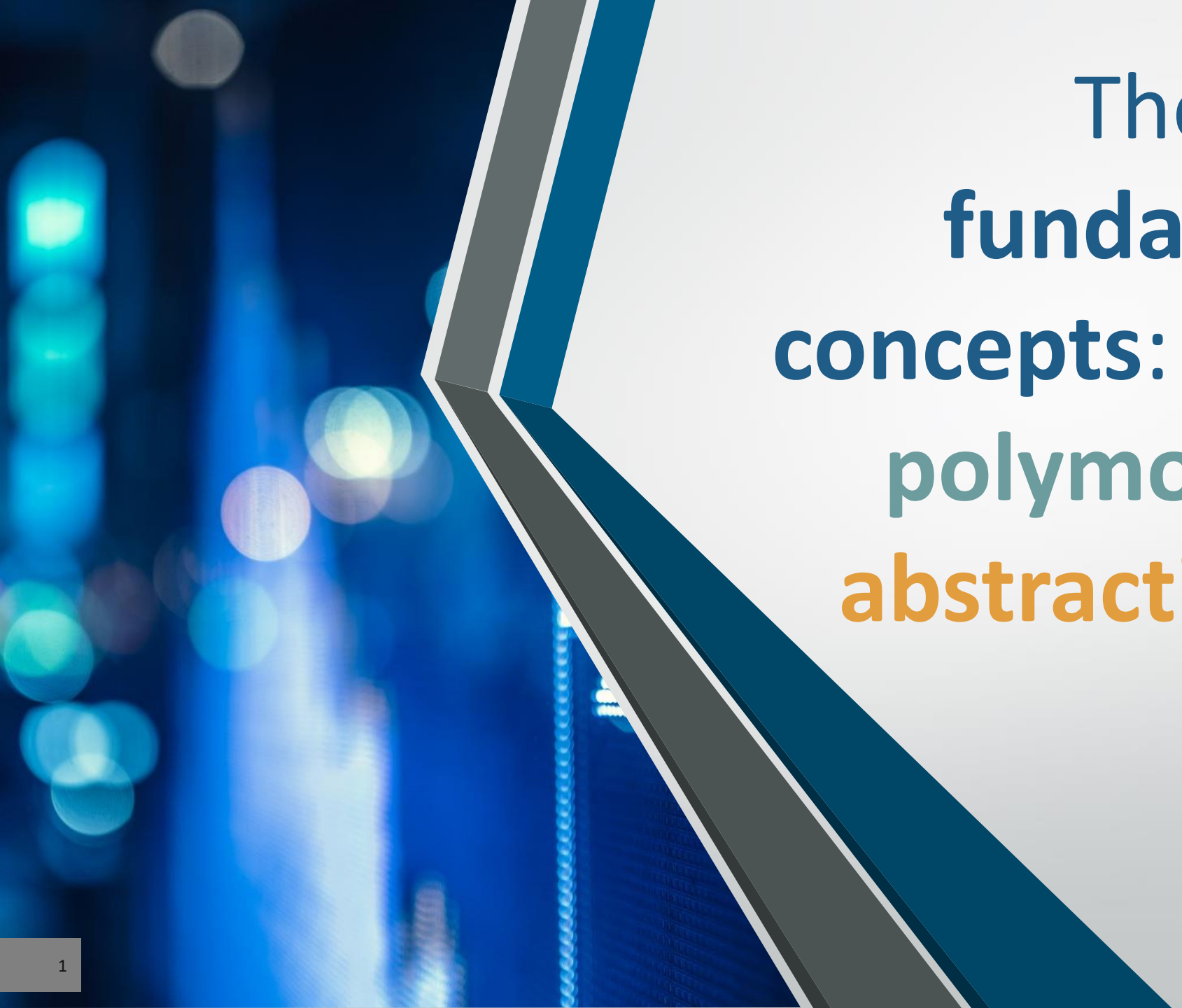


The other three **fundamental OOP concepts**: **inheritance**, **polymorphism**, and **abstraction** (abstract classes and interphases)

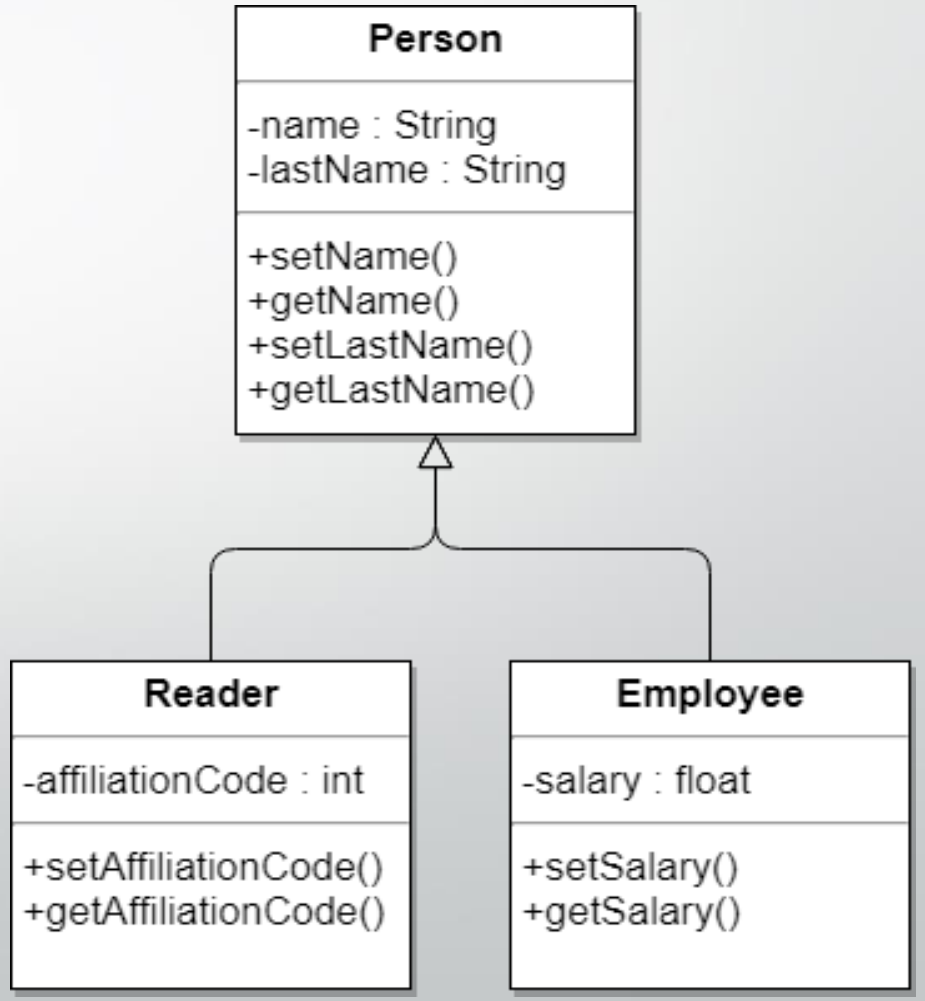
Héritage

L'héritage est un mécanisme qui **facilite la réutilisation du code et la gestion de son évolution**. Elle définit une relation entre deux classes :

- la classe mère (ou super-classe)
- la classe fille (ou sous-classe) qui hérite d'attributs / méthodes de la classe mère.

En Java, une classe (fille) ne peut hériter que d'une seule classe mère (héritage simple). Une classe mère peut par contre avoir plusieurs classes filles. Par défaut toute classe hérite de la classe la plus générale: `Object`.

Diagramme de classes



But what is the point of inheritance in concrete terms?

CHECKING ACCOUNT	SAVINGS ACCOUNT	CERTIFICATE OF DEPOSIT
Account : 123-456	Account : 333-111	Account : 975-579
Balance : $999	Balance : $500	Balance : $12,000
Limit : $9,000	Transfers : 3/6	Expires : 1-1-2020

L'héritage permet donc:

La **réutilisation**: une sous-classe possède les attributs / méthodes de la classe mère (sauf ceux déclarés **`private`**).

La **spécialisation**: on peut aussi redéfinir certains attributs / méthodes de la classe mère dans la sous-classe.

L'extension: on peut ajouter de nouveaux attributs/méthodes propres à la sous-classe.

Option 1

```
class BankAccount {
   int      acctType;
   String   acctNumber;
   double   balance;
   double   limit;
   int      transfers;
   Date     expiry;
}
```

However, both options have more disadvantages than advantages!

Option 2

```
class Checking {
   String acctNumber;
   double balance;
   int bankCode;
   double limit;
}
```

```
class Savings {
   String acctNumber;
   double balance;
   int bankCode;
   int transfers;
}
```

```
class COD {
   String acctNumber;
   double balance;
   int bankCode;
   Date expiry;
}
```

```
class BankAccount {
   String acctNumber;
   double balance;
}
```

Option 3: Heritage

```
class Checking extends BankAccount{
   double limit;
}
```

```
class Savings extends BankAccount{
   int transfers;
}
```

```
class COD extends BankAccount{
   Date expiry;
}
```

Inheritance declaration

La déclaration en Java se fait avec le mot clé **extends**, comme suit:

```
[<protection>] class <classe> extends <super classe>{
   … (re)définition d'attributs …
   … (re)définition de méthodes …
}
```

Remarques :

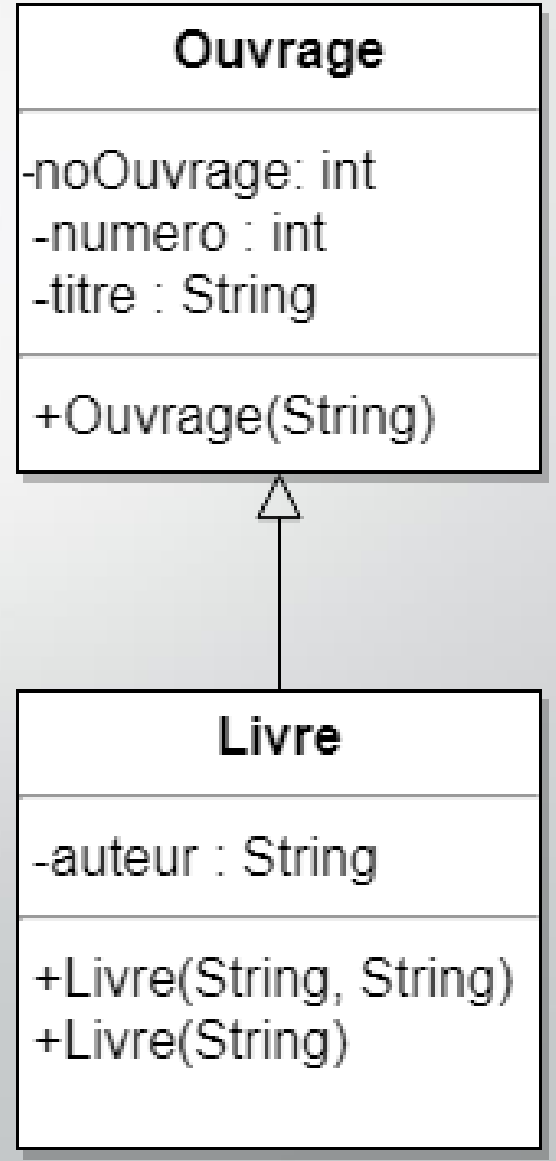
On n'hérite pas des constructeurs de la classe mère ! Si on en a besoin il faut redéfinir ce constructeur dans la classe fille.

Par défaut le constructeur implicite (sans argument) de la classe mère est appelé (il doit donc exister). On peut toutefois appeler un constructeur de la classe mère en utilisant : **super**(<arguments>) . Le mot-clé **super** permettant de référencer les membres de la classe mère.

Inheritance (example)

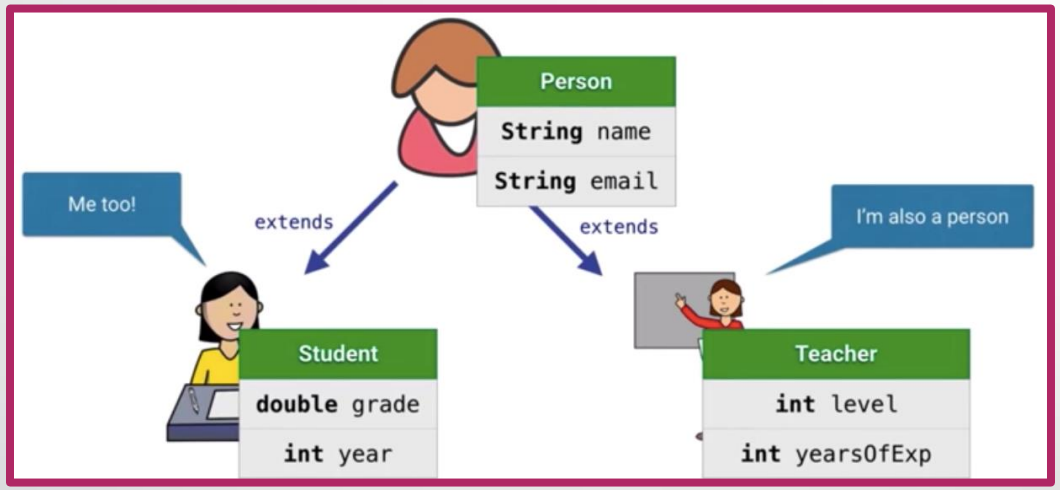
```
public class Ouvrage {
  private static int noOuvrage = 0;
  protected int numero;
  protected String titre;

  public Ouvrage(String titre) {
    numero = ++noOuvrage; //numéro séquent.
    this.titre = titre;
  }
}
```

```
public class Livre extends Ouvrage {
  private String auteur;

  public Livre(String titre, String auteur) {
    super(titre);  // doit être la 1ère instruction
    this.auteur = auteur;
  }

  public Livre(String titre) {
    this(titre, null); // on appelle le 1er construc
  }
}
```



Polymorphism

Polymorphism is **the ability of an object to take many forms**. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.



Both, Teacher and Student are extending the class Person.

So, they are by default of type Person as well.

We can declare both Student and Teacher as of type Person and still initialize them using their own constructor

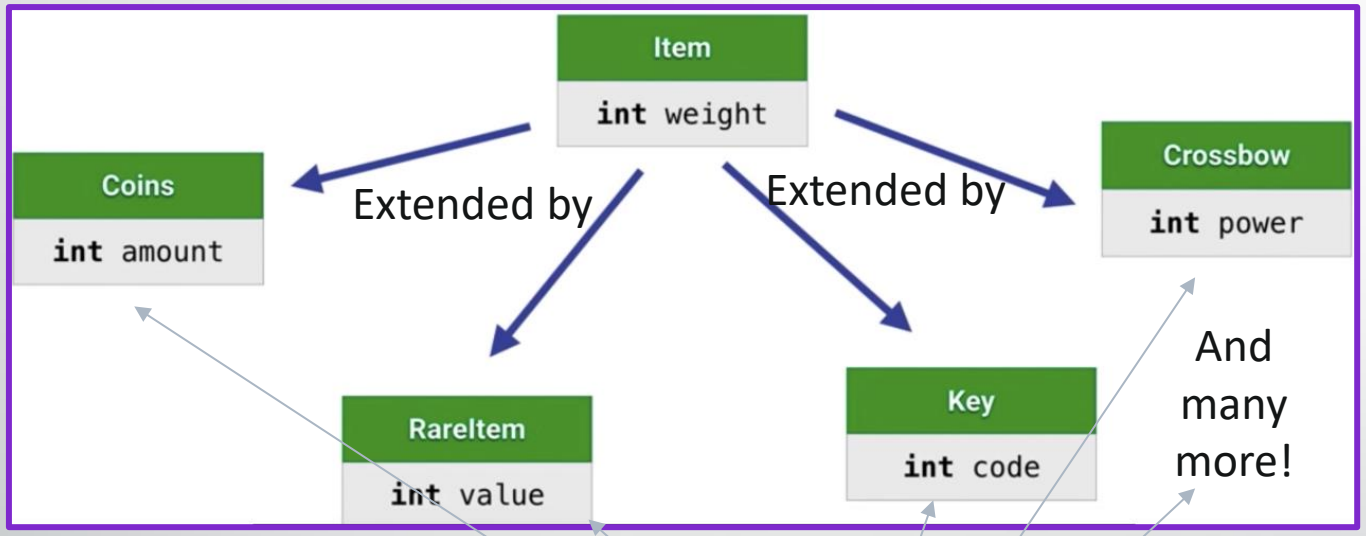
```
Student student = new Student();
Teacher teacher = new Teacher();
```

```
Person student = new Student();
Person teacher = new Teacher();
```

OK, but is that really helpful?

YES! Let's look at this scenario: Imagine you are working with a team of Java developers building a game that involves a bag of items that they can carry with them. However, the bag can only carry a total weight of 20 kilograms.

And you are the responsible for implementing the logic that checks if someone can add another item to the bag or not! So far, the team created a lot of items they want to take:



Which ones in the bag?

A single method that can be used for any objects that behaves as an Item

```
public class Bag{

    int currentWeight;
    boolean canAddItem(Item item);

}
```

And then

```
boolean canAddItem(Item item){
    if(currentWeight + item.weight > 20){
        return false;
    }
    else{
        return true;
    }
}
```

QUIZ: POLYMORPHISM

You have a class **Book** that has the fields `title` and `numberOfPages`, as well as 2 more classes **`Novel`** and **`TextBook`** (with their own fields) that extend class Book.

Which of the following statements is/are correct?
(There could be more than 1 correct answer)

1. Book someBook = new Book();
2. Book someBook = new Novel();
3. Novel someNovel = new Book();
4. Novel someBook = new TextBook();

Typage statique : l'opérateur `cast`

Supposons **`c`** déclaré comme un objet de classe **`C`**.

```
C c;
```

On peut évidemment lui affecter n'importe quel objet de même type **`C`** (ex: par **`new C()`**). Mais pour exploiter l'héritage on peut aussi vouloir lui affecter un objet **`d`** de type **`D`** qui est :

1. **plus spécialisé** (c-à-d D est une sous-classe de C). La conversion de type est alors implicite, on peut simplement écrire: **`c = d;`**
2. **plus général** (c-à-d D est une super-classe de C). La conversion doit alors être explicite avec l'opérateur de cast. **`c = (C) d;`**

En conclusion on peut affecter un objet à un objet « plus général ». Pour faire l'inverse on doit utiliser un cast.

Remarque: On ne peut pas faire un cast s'il n'y a pas de lien d'héritage entre C et D.

Utilisation de `cast`: erreur de compilation

```
class A {
  int x = 1;
}
```

```
class B extends A
{
  int y = 2;
}
```

`A a = new B();` cast inutile: `B` est une sous-classe de `A`

…

`int z = a.y;` erreur à la compilation

Le compilateur émet une erreur (**`cannot find variable y`**) car la variable **`y`** n'est pas définie dans **`A`** (ni dans une de ses super-classes). Bien que nous sachions que **`a`** est en fait du type **`B`**, le compilateur ne le devine pas. On utilise le cast:

```
int z = ((B) a).y;
```

ATT: l'opérateur `.` est plus prioritaire que l'opérateur `(cast)` il faut donc utiliser des parenthèses pour forcer l'ordre d'évaluation.

Utilisation de cast: erreur à l'exécution

```
class A {
  int x = 1;
}
```

```
class B extends
A {
  int y = 2;
}
```

```
A a = new A();
…
int z = ((B) a).y;
```

erreur à l'exécution

Déclenche une exception (`java.lang.ClassCastException: A`) pour signaler l'erreur. En effet, `a` est de type `A` et ne peut être converti en `B` car une classe mère ne peut pas être converti en fille (le contraire oui).

Remarque: cette erreur n'est pas détectée à la compilation.

Résolution des variables

La **résolution des variables est faite par typage statique**. On détermine donc au moment de la compilation la variable utilisée.

Supposons une classe A super-classe de B, et B super-classe de C.
Si on a un objet typé comme C, alors on cherchera la définition de la variable dans C, puis dans B, puis dans A.

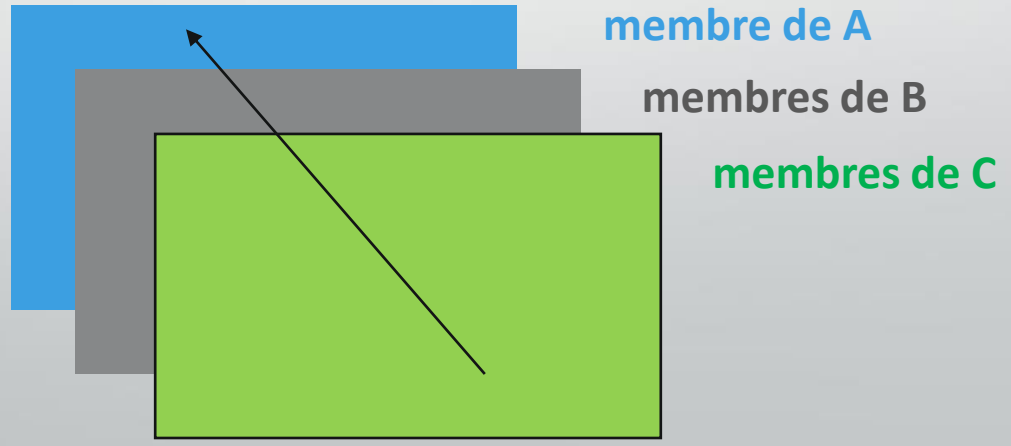
```
C obj = new C();
```

Si le même objet est typé B on cherchera dans B puis dans A.

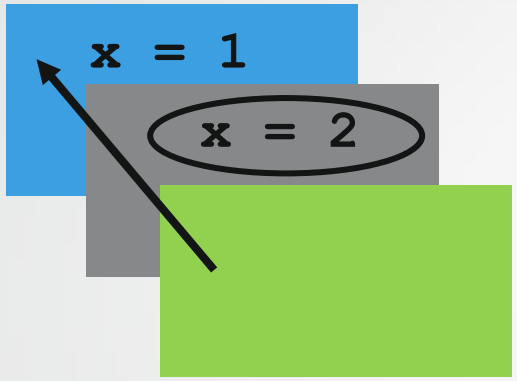
```
B obj = new C();
```

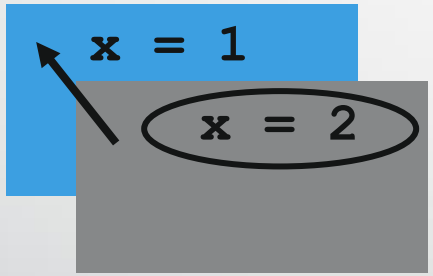
Si le même objet est typé A on cherchera dans A.

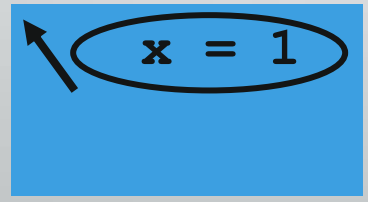
```
A obj = new C();
```



Résolution des variables








```
class A {
  int x = 1;
}

class B extends A {
  int x = 2;
}

class C extends B {
  …
}

C c = new C();
System.out.println("x = " + c.x);
B b = c;
System.out.println("x = " + b.x);
A a = c;
System.out.println("x = " + a.x);
```

Affichera ?

```
x = 2
x = 2
x = 1
```

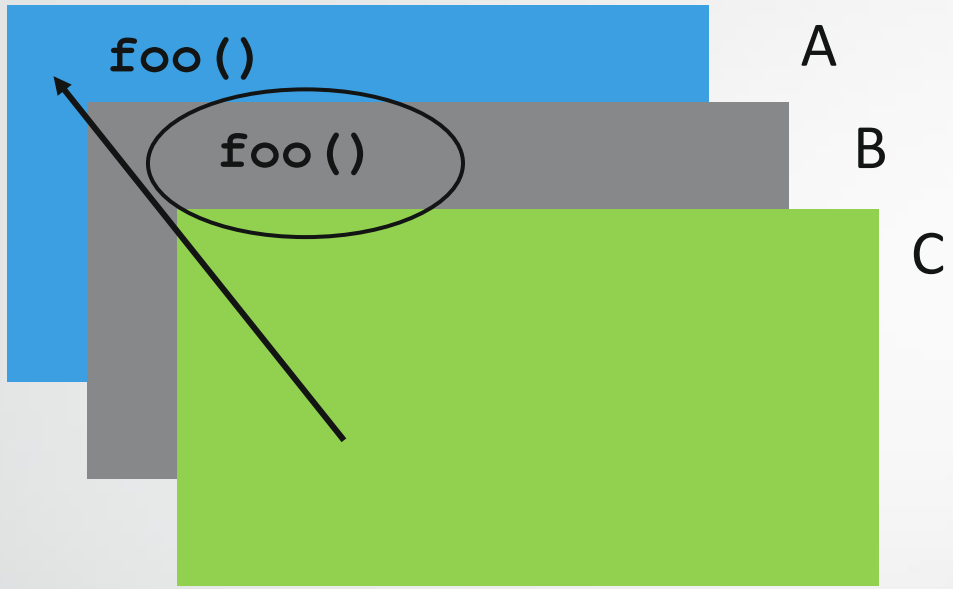
Résolution des méthodes

La résolution des appels de méthodes est faite par le mécanisme de **liaison dynamique**. **On détermine au moment de l'exécution la méthode à appeler (c'est donc différent du traitement des variables!)**.

L'idée de la liaison dynamique (**ou liaison retardée – *late binding***) est de déterminer quel est le vrai type d'un objet pour déterminer la méthode à appeler. Ceci ne peut se faire que dynamiquement à l'exécution.

Ce mécanisme se justifie par le fait qu'on veut généralement qu'un objet conserve ses comportements même si on le stocke dans un objet plus général (et donc on voudrait que ce soit ses méthodes propres qui soient invoquées). De cette manière ça marche donc comme on s'y attend !

Résolution des méthodes




```
class A {
  void foo() {
    System.out.print("foo A");
  }
}

class B extends A {
  void foo() {
    System.out.print("foo B");
  }
}

class C extends B {
  …
}

C c = new C();
c.foo();
B b = c;
b.foo();
A a = c;
a.foo();
```

Avec cette exemple le comportement est **similaire à celui de la résolution des variables car on est à l'extérieur des classes**, et donc il affichera:

```
foo B
foo B
foo A
```

QUIZ QUESTION. Quelle es la réponse de c.test()?

```
public class Main {
   public static void main(String[] args) { C c = new C(); c.test(); }
}
```

```
public class A {
  int x = 1;
  void m(){
    System.out.println("M from A");
  }
}
```

```
class B extends A{
  int x = 2;
  void m(){
    System.out.println("M from B");
  }
}
```

```
class C extends B{
  int x = 3,a;
  void m(){
    System.out.println("M from C");
  }
  void test(){
    a = super.x;
    System.out.println(a);
    a = ((B)this).x;
    System.out.println(a);
    a = ((A)this).x;
    System.out.println(a);
    this.m();
    super.m();
    ((B) this).m();
  }
}
```

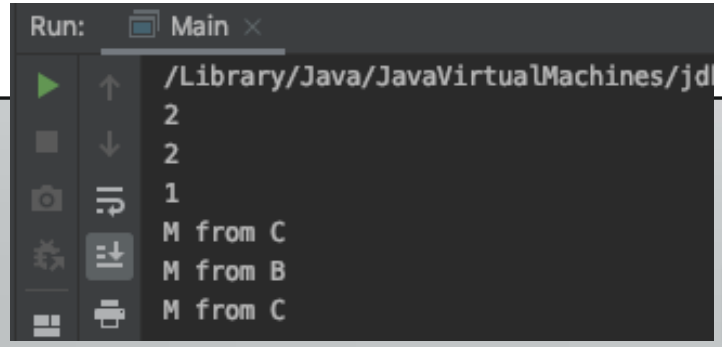
QUIZ QUESTION. Quelle es la réponse de c.test()?

```
public class Main {
   public static void main(String[] args) { C c = new C(); c.test(); }
}
```

```
public class A {
  int x = 1;
  void m(){
    System.out.println("M from A");
  }
}
```

```
class B extends A{
  int x = 2;
  void m(){
    System.out.println("M from B");
  }
}
```

```
class C extends B{
  int x = 3,a;
  void m(){
    System.out.println("M from C");
  }
  void test(){
    a = super.x;
    System.out.println(a);
    a = ((B)this).x;
    System.out.println(a);
    a = ((A)this).x;
    System.out.println(a);
    this.m();
    super.m();
    ((B) this).m();
  }
}
```



Why?

Overriding Java methods: the Object class

Toute classe hérite (directement ou indirectement) de la classe `Object`. Voici quelques méthodes qu'offre cette classe :

`String toString()` : retourne une chaîne associé à l'objet. Cette méthode peut être appelée explicitement. Elle est invoquée implicitement à chaque fois que l'objet est passé et qu'une chaîne est attendue. Par exemple dans `System.out.println`. Peut bien sûr être redéfinie.

`boolean equals(Object obj)` : retourne vrai que si l'objet est le même que `obj`. Cette méthode teste `this == obj` (même adresse) et doit donc généralement être redéfinie.

`void finalize()` : est exécuté juste avant la libération de l'objet par le Garbage Collector. Peut-être redéfinie.

`Class getClass()` : retourne la classe réelle de l'objet (introspection). Peut être utilisé dans une redéfinition de `equals`.

Overriding the "toString" method

```
public class Lecteur {
  protected String nom;
  protected String prenom;
  …
  public String toString() {
    return "lecteur: nom = " + nom +
      ", prénom = " + prenom;
  }
}

…
Lecteur x = new Lecteur("Deschamps", "Antoine");
System.out.println(x);
```

Affichera :

```
lecteur: nom = Deschamps, prénom = Antoine
```

Overriding the "equals" method

```
public class Lecteur {
  protected String nom;
  protected String prenom;
  …
  public boolean equals(Object obj) {
    if (obj == null || !(obj instanceof Lecteur))
      return false;
    Lecteur l = (Lecteur) obj;
    return nom.equals(l.nom) && prenom.equals(l.prenom);
  }
}
```

```
Lecteur x = new Lecteur("Deschamps", "Antoine");
Lecteur y = new Lecteur("Deschamps", "Antoine");
if (x == y)
```
on a vu que vaut faux !

```
if (x.equals(y))
```
mais `equals` vaut vrai !

Classes et méthodes abstraites

On a vu qu'une classe contient la définition d'attributs et de méthodes.

On peut **vouloir déclarer une méthode sans en donner son implémentation. Ceci dans le but de forcer toute classe héritière à implémenter cette méthode**. On parle alors de méthode abstraite et par extension de classe abstraite.

Une **méthode abstraite** est donc une méthode pour laquelle on ne définit pas son implémentation (le corps de la méthode est absent).

Une classe comportant au moins une méthode abstraite est dite **classe abstraite**.

```
abstract class Animal {
   abstract void soundOfAnimal();  // It is just an idea
}
class Cat extends Animal {
   void soundOfAnimal() {
      System.out.println("Meoh");
      //Implementation of the idea according to requirements of sub class
   }
}
class Dog extends Animal{
   void soundOfAnimal() {
      System.out.println("Bow Bow");
      //Implementation of the idea according to requirements of sub class
   }
}
```

Classes et méthodes abstraites

On utilise le même mot-clé **`abstract`** pour **déclarer une classe abstraite et une méthode abstraite**. Exemple :

```
abstract public class Polygone {
  …
  public double perimetre() {
    …
    return …;
  }

  abstract public double surface();
}
```

Remarque: **une classe abstraite *ne peut pas être instanciée!***

`Polygone p = new Polygone();` Erreur de compilation

Elle ne sert à rien alors?

Classes et méthodes abstraites

La seule utilisation d'une classe abstraite est donc l'héritage!

Lorsqu'une classe hérite d'une classe abstraite elle doit fournir une implémentation pour toutes les méthodes abstraites de la classe abstraite (les autres méthodes peuvent si besoin être redéfinies mais cela n'est évidemment pas obligatoire). Exemple :

```
abstract public class Polygone {
  …
  public double perimetre() {
    …
    return …;
  }

  abstract public double surface();
}
```

```
public class Rectangle extends Polygone {
  protected Point sg, sd, id, ig;

  …
  public double surface() {
    double l1 = sg.distance(sd);
    double l2 = sg.distance(ig);
    return l1 * l2;
  }
}
```

Classes abstraites : utilité

Les intérêts d'une classe abstraite sont :

1. de permettre de **factoriser le code commun** (qui est implémenté) tout en laissant l'implémentation de ce qui est spécifique (ie. Ce qui diverge) aux classes filles.
2. d'**obliger les classes qui héritent à fournir une implémentation** pour les méthodes abstraites.
3. **spécifier la signature de la méthode/classe** (méthode/classe abstraite). C-à-d, pour savoir ce qu'il faut pour l'utiliser (p.ex. ce qu'on doit fournir à la méthode via les arguments et ce que l'on récupère via la valeur de retour). On dit qu'on a « abstrait » ces fonctionnalités spécifiques.

Interfaces

La notion d'interface est absolument centrale en Java, et massivement utilisée dans le design des API du JDK et de Java EE. **Tout bon développeur Java doit absolument maîtriser ce point parfaitement!**

La notion d'interface est profondément modifiée en Java 8. Jusqu'en Java 7 une interface ne peut posséder que des méthodes abstraites (donc des signatures de méthodes) et des constantes.

À partir de Java 8, on peut ajouter deux éléments supplémentaires dans une interfaces : **des méthodes statiques et des méthodes par défaut**.

Les interfaces sont aussi utilisées afin de faciliter la maintenance du code:
*On peut imaginer une classe Transport, puis des classes Avion, Voiture, Moto, Camion, Velo, etc... qui héritent de Transport. Certains de ces classes ont besoin de faire le plein. Pour cela elles se rendent dans une station service pour faire le plain, mais seulement celles qui implémentent l'**interface** contenant la méthode faiseLePlein(...) (Avion, Voiture…) pourront faire le plain (pas les vélos, tricycles… qui n'implémentent pas l'interface)*

Interfaces

```
public  class Transport {
   public  void roule() ;
}

public  class Voiture  extends Transport {
   public  void conduit() ;
}

public  class Avion  extends Transport {
   public  void vole() ;
}

public  class Moto  extends Transport {
   public  void seFaufile() ;
}

public  class Velo  extends Transport {
   public  void pedale() ;
}
```

Codons à présent notre station service avec sa méthode **`ajouterCarburant(...)`**.
Malheureusement, dans la hiérarchie de **`Transport`**, il y a la classe **`Velo`**, et un vélo ne fait pas le plein!

```
public  class StationService {
   public  void ajouterCarburant(Transport transport) {
      if (transport  instanceof Velo) {
         // ne pas faire le plein
      }  else {
         // faire le plein pour chaque moyen de locomotion
      }
   }
}
```

Cette méthode fonctionne dans notre cas, mais si on ajoute d'autres classes (ex. **`Tricycle`**) ou si elle est étendue par des classes dont on n'aura pas de connaissance, elle ne fonctionne pas!
Le principal problème de cette approche est qu'il faut modifier le code de cette méthode à chaque fois que l'on ajoute des classes dans la hiérarchie de **`Transport`**.

C'est là que les interfaces entrent en jeu et nous aident à résoudre notre problème.

Écrivons une interface Motorise, et utilisons-la dans notre hiérarchie d'objets.

```
public  interface Motorise {  // notre interface
   public  void faisLePlein() ;
}

public  class Transport {
   // une instance de Transport ne sait pas toujours faire le plein
   public  void roule() {}
}

public  class Voiture  extends Transport  implements Motorise {
   public  void conduit() {}
   public  void faisLePlein() {}
}

public  class Avion  extends Transport  implements Motorise {
   public  void vole() ;
   public  void faisLePlein() {}
}

public  class Moto  extends Transport  implements Motorise {
   public  void seFaufile() ;
   public  void faisLePlein() {}
}

public  class Velo  extends Transport {  // ne sait pas faire le plein
   public  void pedale() ;
}
```

On peut alors écrire notre classe **`StationService`** de la façon suivante

```
public  class StationService {
   public  void ajouterCarburant(Motorise mot) {
       mot.faisLePlein() ;
    }
}
```

Ce qui nous donne un code qui ne dépend plus des classes de la hiérarchie de **`Transport`**, et en particulier du fait que l'on peut en ajouter dedans. Ici, notre station service accepte toute instance d'une classe qui possède une méthode **`faisLePlein()`**, dont l'existence est spécifiée par l'interface **`Motorise`**.

C'est bien sûr ce type d'approche qu'il faut choisir lorsque l'on veut traiter des cas analogues à celui-là!

```
public  interface Chargeable {  //interface pour les trans chargble
   public  void charger() ;
}
public  interface Motorise {  //interface  pour les transp à  moteur
   public  void faisLePlein() ;
}
public  class Transport {
   public  void roule() {}
}
public  class Voiture  extends Transport  implements Motorise,
Chargeable {
   public  void conduit() {}
   public  void faisLePlein() {}
   public  void charger() {}
}
public  class Avion  extends Transport  implements Motorise,
Chargeable {
   public  void vole() ;
   public  void faisLePlein() {}
   public  void charger() {}
}
public  class Moto  extends Transport  implements Motorise {
   public  void seFaufile() ;
   public  void faisLePlein() {}
}
public  class Velo  extends Transport {
   public  void pedale() ;
}
```

Avec **l'interface `Chargeable`**, on peut aussi considérer (voir) certains des objets de type `Transport` comme des transports de charge et les utiliser par exemple dans un **entrepôt pour charger des marchandises sans avoir besoin de demander de quel type est chaque objet et sans devoir changer le code d'`Entrepot` à chaque fois qu'un nouveau moyen de transport est créé dans le système!**

```
public  class Entrepot {
  public  void chargerMarchandise(Chargeable ch){
      ch.charger() ;
  }
}

public  class StationService {
   public  void ajouterCarburant(Motorise mot) {
      mot.faisLePlein() ;
   }
}
```

Synthèse technique sur les interfaces

Une **interface** est assimilable à une classe abstraite ne contenant que des déclarations de méthodes abstraites. Il est alors inutile de spécifier les **`abstract`**.

Une interface ne peut pas contenir de déclarations de variables mais on peut déclarer des constants (les mot clés **`static final`** sont alors optionnels).

Les méthodes sont toujours **`public`** (le mot-clé est donc optionnel).

Les méthodes peuvent être définis par défaut (**`default`**). Une méthode par défaut permet d'écrire une méthode dans une interface, en fixant sa signature et en donnant une implémentation.

Une interface peut **étendre** plusieurs autres interfaces (c-à-d inclure les fonctionnalités prévues par d'autres interfaces). On utilise le mot clé **`extends`** comme pour l'héritage mais ici il peut être multiple (on sépare par des virgules toutes les interfaces que l'on reprend).

Interface ou classe abstraite ? Les deux !

L'interface est un « **contrat** » que s'engagent à respecter l'utilisateur et l'implémenteur. On l'utilise pour décrire une API : un ensemble de fonctionnalités offertes à l'utilisateur (sous la forme d'un ensemble de classes ou d'un package). Celui-ci ne se soucie pas de l'implémentation (sauf à titre informatif pour choisir l'implémentation la plus adaptée). **Lorsqu'un ensemble de fonctionnalités est prévu pour un « utilisateur externe » à l'ensemble de fonctionnalités, une interface est la bienvenue!**

La **classe abstraite sert à bien architecturer le code** (i.e. les classes implémentant les fonctionnalités). Cela relève plus de l'implémentation. Son rôle principal est de permettre la **factorisation de code**.

Une classe abstraite est plus structurée et peut fournir des impléms par défaut. **Une interface est plus souple car une classe peut implémenter plusieurs interfaces (alors qu'elle ne peut hériter que d'une seule classe).**

En Java 8: les interfaces peuvent aussi fournir des implémentations par défaut (mot-clé `default`).

https://www.amazon.de/dp/B0FYZ1NPGD

Classes abstraites	Interfaces
Une classe est automatiquement abstraite si une de ses méthodes est abstraite	Une interface ce n'est qu'une signature avec des méthodes abstraites et des constants (static, final)
On déclare qu'une classe est abstraite par le mot clef abstract: `public abstract class B {}`	Pour définir une interface: `public interface C {}`
Une classe abstraite n'est pas instanciable (on ne peut pas utiliser les constructeurs d' une classe abstraite -> pas de création d'objet de cette classe).	Les sous-classes d'une super-classe qui implémente une interface, héritent (et peuvent redéfinir) des méthodes implémentées.
Une classe qui hérite d'une classe abstraite ne devient concrète que si elle implémente toutes les méthodes abstraites de la classe dont elle hérite.	Une classe qui implémente une interface doit définir le corps de toutes ses méthodes abstraites.
Une méthode abstraite ne contient pas de corps, mais doit être implémentée dans les sous-classes non abstraites: abstract nomDeMéthode (<arguments>); ce n'est qu'une signature	Les méthodes sont explicitement ou implicitement abstraites : [<modificateur>] [abstract] <type> <nomMéthode> ([<paramètres formels>]);
Une classe est abstraite **peut contenir des méthodes non abstraites et des déclarations de variables ordinaires**.	Une interface ne comporte **que des constantes et des méthodes abstraites**. Les constantes sont explicitement ou implicitement static et final.
Une classe ne peut étendre qu'une seule classe abstraite: `class A extends B {}`	Une classe peut implémenter plus d'une interface: `class A implements C, D, E {}`