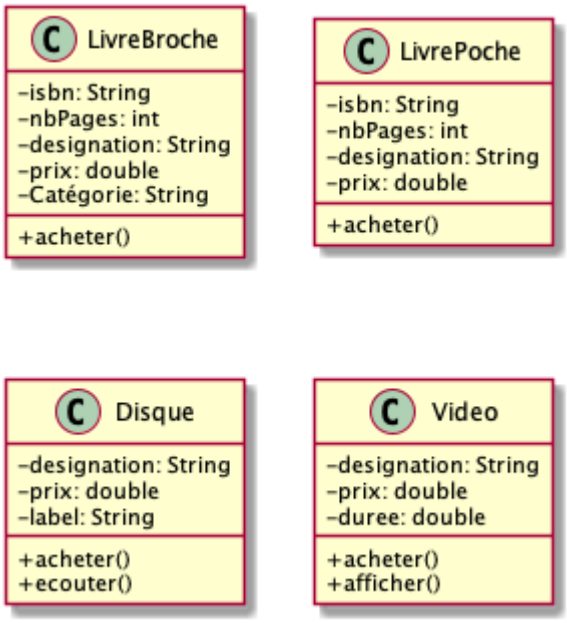
Développement orienté objet

Héritage

Motivation



- 4 classes différentes mais des propriétés communes
- Solution : définir une classe générale et en déduire des classes qui la spécialisent

Héritage de classe

On peut définir une classe **héritant** d'une autre classe.

- La classe héritante (ou **enfant**)
  - **récupère tous** les comportements (accessibles) de la classe dont elle hérite,
  - peut **modifier** certains comportements hérités,
  - peut **ajouter** de nouveaux comportements qui lui sont propres.
- lorsqu'une classe hérite d'une autre en programmation orientée objet, une instance de la classe héritante (la classe dérivée) est aussi considérée comme une instance de la classe dont elle hérite (la classe parente). On parle de **polymorphisme**
  - on parle de **sous-classe** (donc sous-type)
  - une instance de sous-classe est également du type de la classe mère (ou super-classe)

Interprétation : *est un*

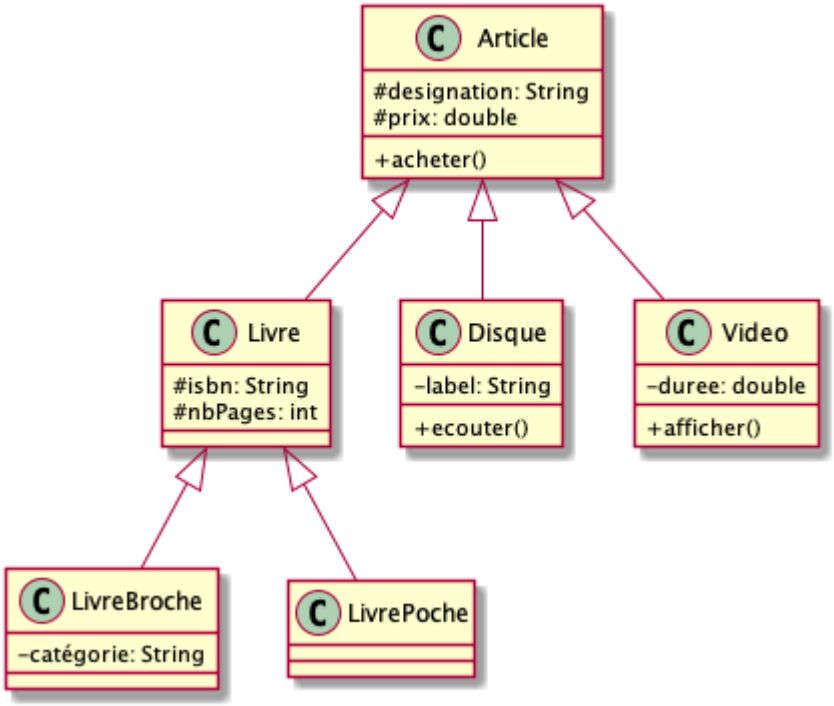
On fait souvent référence à l'héritage comme réalisant la relation *est un*.

- un **livre** est un **article** contenant des pages
- un **livre broché** est un **livre** contenant des pages, dont la couverture est en carton

- un **livre de poche** est un **article** contenant des pages, dont la couverture est en papier
- un **disque** est un **article** de forme ronde, qui contient des enregistrement sonore
- une **vidéo** est un **article** qui contient des images animées

Relation d'héritage en UML

En UML, on représente l'héritage par une flèche (*tête vide*) pointant vers la classe parent.



- Par héritage, un livre broché est aussi un livre et un article
- les attributs / méthodes publiques définie dans **Article** sont aussi dans **LivreBroche**
- idem pour celle de **Livre**
- **Attention :** ce ne serait pas le cas si les attributs / méthodes étaient privées

Nouvelle visibilité : `protégé (protected)`

- `privé (-)`: visibilité d'une propriété limitée à la classe elle-même
- `publique (+)`: aucune limite sur la visibilité
- `protégé (#)` : visibilité limitée à la classe elle-même et à ses sous-classes / classes enfants

Important : Une sous-classe / classe enfant n'a pas accès aux propriétés privées de la classe parent

Héritage simple en TypeScript

On dit qu'une classe `B` hérite de la classe `A` si `B` est une spécialisation de `A`. On dit aussi que `B` est une sous-classe de `A` et que `A` est une super-classe de `B`.

La classe `B` hérite donc de la classe `A` en héritant de ses attributs et de ses méthodes publiques. On dit que la classe `B` étend la classe `A`.

La classe `Personne` ci-dessous est la classe `A`.

```
class Personne {
  private nom: string;
  private prenom: string;
  private age: number;

  constructor(nom: string, prenom: string, age: number) {
    this.nom = nom;
    this.prenom = prenom;
    this.age = age;
  }

  sePresenter(): void {
    console.log(
      `Bonjour, je m'appelle ${this.prenom} ${this.nom} et j'ai ${this.age} ans.`,
    );
  }

  getNom(): void {
    return this.nom;
  }

  getPrenom(): void {
    return this.prenom;
  }

  getAge(): void {
    return this.age;
  }
}
```

On va créer une classe `Etudiant` qui hérite de la classe `Personne`. On va donc créer une classe `Etudiant` qui étend la classe `Personne`.

```
class Etudiant extends Personne {}
```

On peut maintenant créer un objet `etudiant` de type `Etudiant` et appeler la méthode `sePresenter()`.

```
let etudiant = new Etudiant("Dupont", "Jean", 20);
etudiant.sePresenter();
```

```
Bonjour, je m'appelle Jean Dupont et j'ai 20 ans.
```

La classe `Etudiant` hérite donc de la classe `Personne` et peut donc appeler le constructeur de la classe `Personne` et la méthode `sePresenter()`.

Remarque : quand on appelle le constructeur, on appelle le constructeur de la classe `Personne` et non celui de la classe `Etudiant` malgré le fait que l'on écrive `new Etudiant("Dupont", "Jean", 20)`.

Constructeur

On peut modifier le constructeur de la classe `Etudiant` pour qu'il prenne en paramètre le nom, le prénom et l'âge de l'étudiant mais aussi le numéro de l'étudiant.

```
class Etudiant extends Personne {
  private numero: number;

  constructor(nom: string, prenom: string, age: number, numero: number) {
    super(nom, prenom, age);
    this.numero = numero;
  }
}
```

On appelle le constructeur de la classe `Personne` avec `super(nom, prenom, age)`.

Méthodes

On peut ajouter une méthode `getNumero()` à la classe `Etudiant`.

```
class Etudiant extends Personne {
  private numero: number;

  constructor(nom: string, prenom: string, age: number, numero: number) {
    super(nom, prenom, age);
    this.numero = numero;
  }

  getNumero(): number {
    return this.numero;
  }
}
```

On peut maintenant appeler la méthode `getNumero()`.

```
let etudiant = new Etudiant("Dupont", "Jean", 20, 123456);
console.log(etudiant.getNumero());
```

Attributs

On peut ajouter un attribut `adresse` à la classe `Etudiant`.

```
class Etudiant extends Personne {
  private numero: number;
  private adresse: string;

  constructor(
    nom: string,
    prenom: string,
    age: number,
    numero: number,
    adresse: string,
  ) {
    super(nom, prenom, age);
    this.numero = numero;
    this.adresse = adresse;
  }

  getNumero(): number {
    return this.numero;
  }

  getAdresse(): string {
    return this.adresse;
  }
}
```

On peut maintenant appeler la méthode `getAdresse()`.

```
let etudiant = new Etudiant("Dupont", "Jean", 20, 123456, "1 rue de la paix");
console.log(etudiant.getAdresse());
```

Redéfinition de méthodes

On peut redéfinir une méthode de la classe `Personne` dans la classe `Etudiant`.

```
class Etudiant extends Personne {
  private numero: number;
  private adresse: string;

  constructor(
    nom: string,
    prenom: string,
    age: number,
    numero: number,
    adresse: string,
  ) {
    super(nom, prenom, age);
    this.numero = numero;
```

```
    this.adresse = adresse;
  }

  getNumero(): number {
    return this.numero;
  }

  getAdresse(): string {
    return this.adresse;
  }

  sePresenter(): void {
    console.log(
      `Bonjour, je m'appelle ${this.prenom} ${this.nom} et j'ai ${this.age} ans. Je
suis étudiant.`,
    );
  }
}
```

On peut maintenant appeler la méthode `sePresenter()`.

```
let etudiant = new Etudiant("Dupont", "Jean", 20, 123456, "1 rue de la paix");
etudiant.sePresenter();
```

On peut aussi appeler la méthode `sePresenter()` de la classe `Personne` dans la méthode `sePresenter()` de la classe `Etudiant`.

```
class Etudiant extends Personne {
  private numero: number;
  private adresse: string;

  constructor(
    nom: string,
    prenom: string,
    age: number,
    numero: number,
    adresse: string,
  ) {
    super(nom, prenom, age);
    this.numero = numero;
    this.adresse = adresse;
  }

  getNumero(): number {
    return this.numero;
  }

  getAdresse(): string {
    return this.adresse;
```

```
  }

  sePresenter(): void {
    super.sePresenter();
    console.log(`Je suis étudiant.`);
  }
}
```

Cela permet d'éviter de réécrire le code de la méthode `sePresenter()` de la classe `Personne`.

Visibilité des attributs et des méthodes

Un objet de la classe `Etudiant` peut accéder aux attributs et aux méthodes publique de la classe `Personne` car la classe `Etudiant` hérite de la classe `Personne`.

```
console.log(etudiant.getNom());
```

```
Dupont
```

Par contre, il ne peut pas accéder aux attributs et aux méthodes privée de la classe `Personne`.

```
console.log(etudiant.nom);
```

```
error TS2341: Property 'nom' is private and only accessible within class 'Personne'.
```

Il ne peut pas non plus y accéder même quand on est dans la classe `Etudiant`.

```
class Etudiant extends Personne {
  private numero: number;
  private adresse: string;

  constructor(
    nom: string,
    prenom: string,
    age: number,
    numero: number,
    adresse: string,
  ) {
    super(nom, prenom, age);
    this.numero = numero;
    this.adresse = adresse;
  }
```

```
  getNumero(): number {
    return this.numero;
  }

  getAdresse(): string {
    return this.adresse;
  }

  sePresenter(): void {
    super.sePresenter();
    console.log(`Je suis étudiant.`);
    console.log(this.nom);
  }
}
```

```
error TS2341: Property 'nom' is private and only accessible within class 'Personne'.
```

La visibilité `protected` permet d'accéder à un attribut ou à une méthode dans la classe `Personne` et dans les classes qui héritent de la classe `Personne`.

```
class Personne {
  protected nom: string;
  protected prenom: string;
  protected age: number;

  constructor(nom: string, prenom: string, age: number) {
    this.nom = nom;
    this.prenom = prenom;
    this.age = age;
  }

  sePresenter(): void {
    console.log(
      `Bonjour, je m'appelle ${this.prenom} ${this.nom} et j'ai ${this.age} ans.`,
    );
  }
}
```

```
class Etudiant extends Personne {
  private numero: number;
  private adresse: string;

  constructor(
    nom: string,
    prenom: string,
    age: number,
```

```
    numero: number,
    adresse: string,
  ) {
    super(nom, prenom, age);
    this.numero = numero;
    this.adresse = adresse;
  }

  getNumero(): number {
    return this.numero;
  }

  getAdresse(): string {
    return this.adresse;
  }

  sePresenter(): void {
    super.sePresenter();
    console.log(`Je suis étudiant.`);
    console.log(this.nom);
  }
}
```

```
let etudiant = new Etudiant("Dupont", "Jean", 20, 123456, "1 rue de la paix");
etudiant.sePresenter();
```

```
Bonjour, je m'appelle Jean Dupont et j'ai 20 ans.
Je suis étudiant.
Dupont
```

Polymorphisme

Un objet `Etudiant` est aussi considéré comme un objet `Personne`.

```
let personne: Personne = etudiant;
```

On peut donc appeler la méthode `sePresenter()` de la classe `Personne` sur l'objet `personne`.

```
personne.sePresenter();
```

```
Bonjour, je m'appelle Jean Dupont et j'ai 20 ans.
```

Par contre, un objet `Personne` n'est pas considéré comme un objet `Etudiant`.

```
let personne: Personne = new Personne("Dupont", "Jean", 20);
let etudiant: Etudiant = personne;
```

```
error TS2322: Type 'Personne' is not assignable to type 'Etudiant'.
  Types of property 'getNumero' are incompatible.
    Type '() => number' is not assignable to type '() => never'.
      Type 'number' is not assignable to type 'never'.
```

On peut définir une autre classe héritant de la classe `Personne`.

```
class Professeur extends Personne {
  private matiere: string;

  constructor(nom: string, prenom: string, age: number, matiere: string) {
    super(nom, prenom, age);
    this.matiere = matiere;
  }

  getMatiere(): string {
    return this.matiere;
  }

  sePresenter(): void {
    super.sePresenter();
    console.log(`Je suis professeur de ${this.matiere}.`);
  }
}
```

On peut créer un tableau de personnes.

```
let personnes: Personne[] = [];
personnes.push(new Etudiant("Dupont", "Jean", 20, 123456, "1 rue de la paix"));
personnes.push(new Professeur("Durand", "Pierre", 30, "Mathématiques"));
```

On peut parcourir le tableau de personnes et appeler la méthode `sePresenter()` sur chaque personne.

```
for (let personne of personnes) {
  personne.sePresenter();
}
```

```
Bonjour, je m'appelle Jean Dupont et j'ai 20 ans.
Je suis étudiant.
Bonjour, je m'appelle Pierre Durand et j'ai 30 ans.
Je suis professeur de Mathématiques.
```

On peut aussi définir une fonction qui prend en paramètre une personne.

```
function sePresenter(personne: Personne): void {
  personne.sePresenter();
}
```

On peut appeler la fonction `sePresenter()` en passant en paramètre un étudiant.

```
sePresenter(new Etudiant("Dupont", "Jean", 20, 123456, "1 rue de la paix"));
```

```
Bonjour, je m'appelle Jean Dupont et j'ai 20 ans.
Je suis étudiant.
```

On peut appeler la fonction `sePresenter()` en passant en paramètre un professeur.

```
sePresenter(new Professeur("Durand", "Pierre", 30, "Mathématiques"));
```

```
Bonjour, je m'appelle Pierre Durand et j'ai 30 ans.
Je suis professeur de Mathématiques.
```

Héritage chaîné

On peut définir une classe `Enseignant` qui hérite de la classe `Personne`.

```
class Enseignant extends Personne {
  private matiere: string;

  constructor(nom: string, prenom: string, age: number, matiere: string) {
    super(nom, prenom, age);
    this.matiere = matiere;
  }

  getMatiere(): string {
    return this.matiere;
```

```
  }

  sePresenter(): void {
    super.sePresenter();
    console.log(`Je suis enseignant de ${this.matiere}.`);
  }
}
```

On peut définir une classe `Professeur` qui hérite de la classe `Enseignant`.

```
class Professeur extends Enseignant {
  private grade: string;

  constructor(
    nom: string,
    prenom: string,
    age: number,
    matiere: string,
    grade: string,
  ) {
    super(nom, prenom, age, matiere);
    this.grade = grade;
  }

  getGrade(): string {
    return this.grade;
  }

  sePresenter(): void {
    super.sePresenter();
    console.log(
      `Je suis professeur de ${this.matiere} et j'ai le grade ${this.grade}.`,
    );
  }
}
```

Les `Professeur` sont donc des `Enseignant` et ce sont donc aussi des `Personne`. Par exemple, on peut créer un tableau de personnes.

```
let personnes: Personne[] = [];
personnes.push(new Etudiant("Dupont", "Jean", 20, 123456, "1 rue de la paix"));
personnes.push(
  new Professeur(
    "Durand",
    "Pierre",
    30,
    "Mathématiques",
    "Maître de conférences",
```

```
  ),
);
```

On peut parcourir le tableau de personnes et appeler la méthode `sePresenter()` sur chaque personne.

```
for (let personne of personnes) {
  personne.sePresenter();
}
```

```
Bonjour, je m'appelle Jean Dupont et j'ai 20 ans.
Je suis étudiant.
Bonjour, je m'appelle Pierre Durand et j'ai 30 ans.
Je suis enseignant de Mathématiques.
Je suis professeur de Mathématiques et j'ai le grade Maître de conférences.
```

On peut aussi définir une fonction qui prend en paramètre une personne.

```
function sePresenter(personne: Personne): void {
  personne.sePresenter();
}
```

On peut appeler la fonction `sePresenter()` en passant en paramètre un `Professeur`.

```
sePresenter(
  new Professeur(
    "Durand",
    "Pierre",
    30,
    "Mathématiques",
    "Maître de conférences",
  ),
);
```

```
Bonjour, je m'appelle Pierre Durand et j'ai 30 ans.
Je suis enseignant de Mathématiques.
Je suis professeur de Mathématiques et j'ai le grade Maître de conférences.
```

Par contre, si on définit une fonction qui prend en paramètre un `Enseignant`.

```
function sePresenterEnseignant(enseignant: Enseignant): void {
  professeur.sePresenter();
```

```
}
```

On ne peut pas passer en paramètre une `Personne` ou un `Etudiant`.

```
sePresenterEnseignant(new Personne("Durand", "Pierre", 30));
sePresenterEnseignant(
  new Etudiant("Dupont", "Jean", 20, 123456, "1 rue de la paix"),
);
```

```
error TS2345: Argument of type 'Personne' is not assignable to parameter of type
'Enseignant'.
error TS2345: Argument of type 'Etudiant' is not assignable to parameter of type
'Enseignant'.
```