Introduction à la programmation et au développement d'applications en Java

Paramètres : overloading & multiplicité

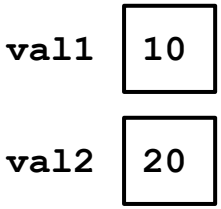
Introduction

- Rappels sur l'immuabilité des paramètres
- Surcharge de constructeurs/méthodes
- Nombre variable de paramètres

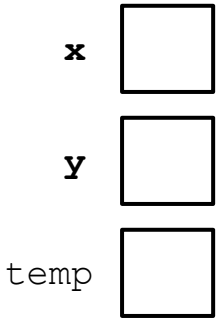
Immuabilité des paramètres

- Lors de l'appel d'une méthode, chaque argument est passé en faisant une copie de la valeur (passage par valeur, *by-value*)
- **Le paramètre qui reçoit la valeur agit donc comme une variable locale à la méthode : tout changement de valeur sur le paramètre n'affectera pas l'argument d'origine**

Exemple avec types primitifs

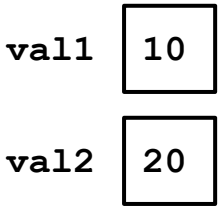



```
int val1 = 10;
int val2 = 20;
// print de val1, val2 affiche 10, 20

echanger(val1, val2);
```

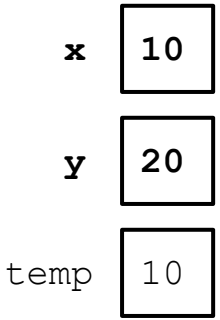



```
void echanger(int x, int y) {
  int temp = x;
  x = y;
  y = temp;

}
```

Exemple avec types primitifs

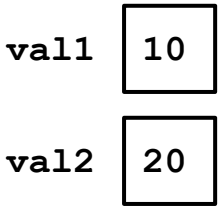



```
int val1 = 10;
int val2 = 20;
// print de val1, val2 affiche 10, 20

echanger(val1, val2);
```

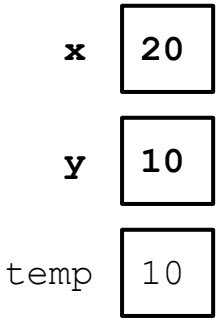



```
void echanger(int x, int y) {
  int temp = x;
  x = y;
  y = temp;

}
```

Exemple avec types primitifs

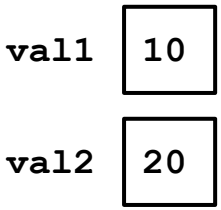



```
int val1 = 10;
int val2 = 20;
// print de val1, val2 affiche 10, 20

echanger(val1, val2);
```

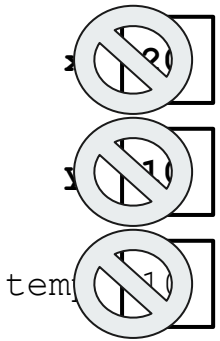



```
void echanger(int x, int y) {
  int temp = x;
  x = y;
  y = temp;
  // print de x, y affiche 20, 10
}
```

Exemple avec types primitifs

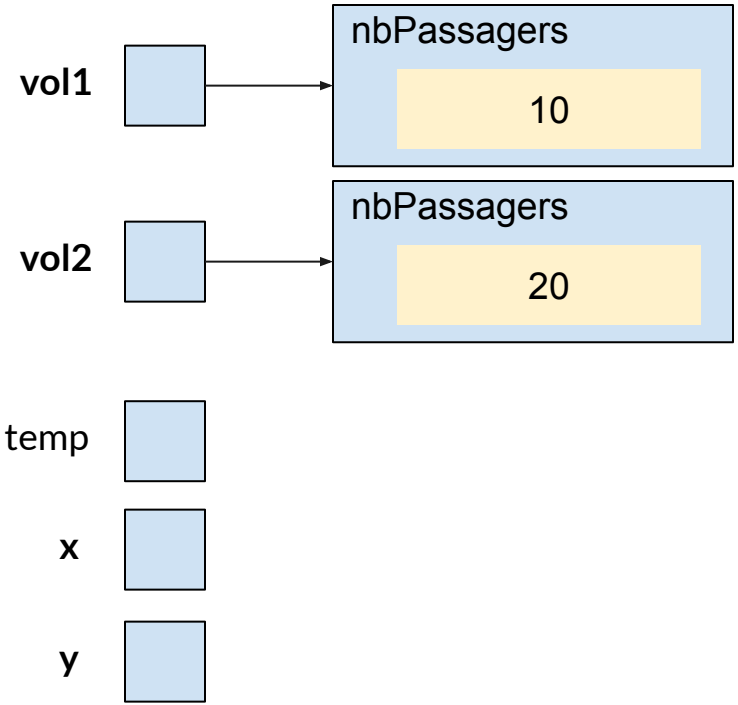



```
int val1 = 10;
int val2 = 20;
// print de val1, val2 affiche 10, 20

echanger(val1, val2);
// print de val1, val2 affiche toujours 10, 20
```



```
void echanger(int x, int y) {
  int temp = x;
  x = y;
  y = temp;
  // print de x, y affiche 20, 10
}
```

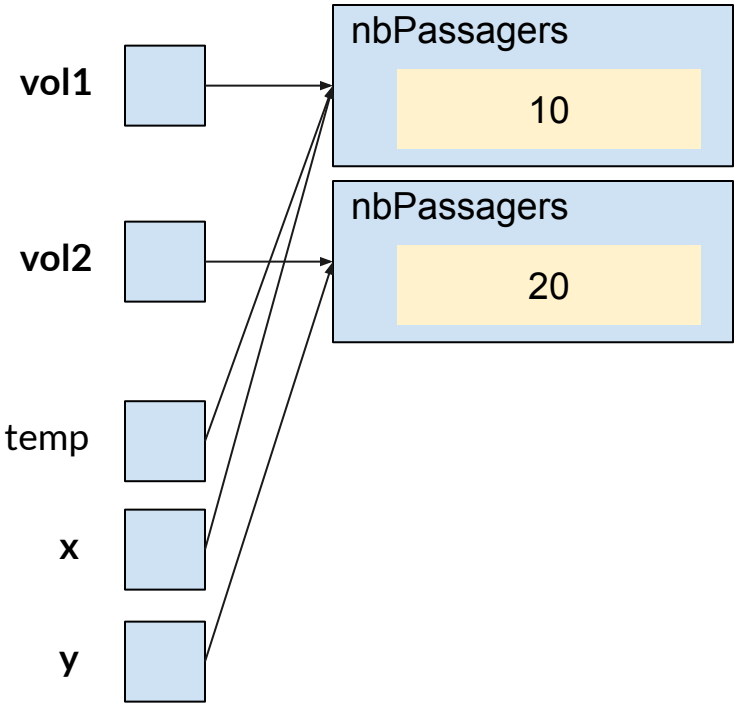
Exemple avec types références




```
Vol vol1 = new Vol(10);
Vol vol2 = new Vol(20);
// print de nbPassagers affiche 10, 20

echanger(vol1, vol2);
```

```
void echanger(Vol x, Vol y) {
  Vol temp = x;
  x = y;
  y = temp;

}
```

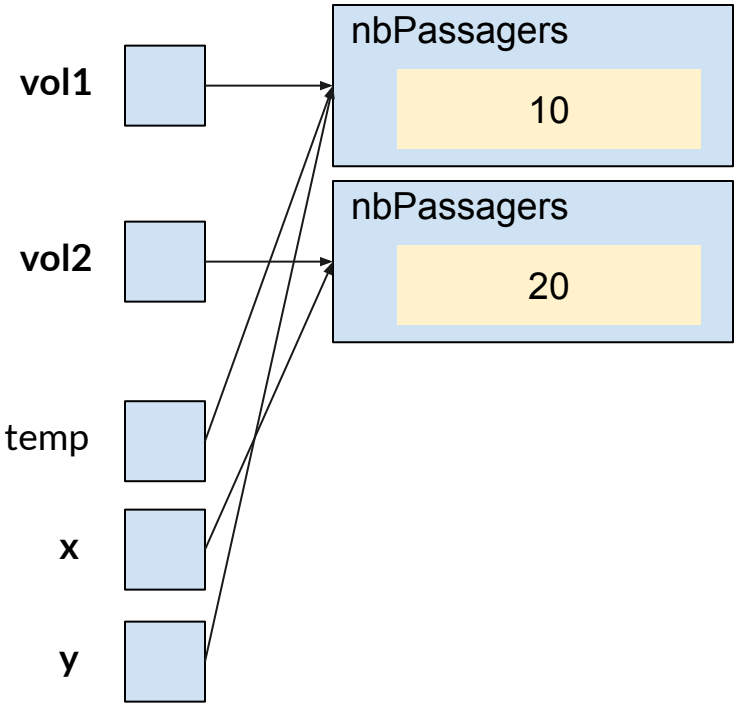
Exemple avec types références




```
Vol vol1 = new Vol(10);
Vol vol2 = new Vol(20);
// print de nbPassagers affiche 10, 20

echanger(vol1, vol2);
```

```
void echanger(Vol x, Vol y) {
  Vol temp = x;
  x = y;
  y = temp;

}
```

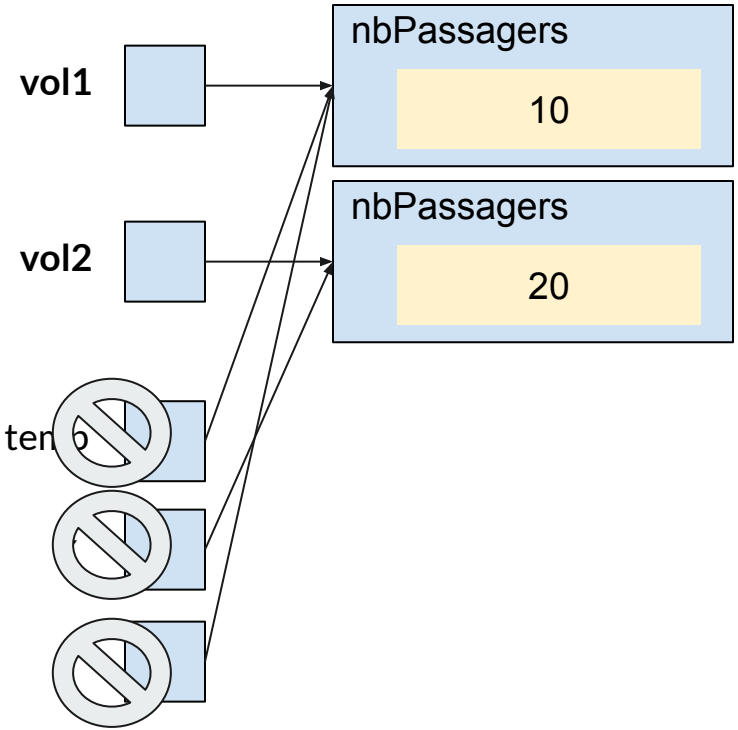
Exemple avec types références




```
Vol vol1 = new Vol(10);
Vol vol2 = new Vol(20);
// print de nbPassagers affiche 10, 20

echanger(vol1, vol2);
```

```
void echanger(Vol x, Vol y) {
  Vol temp = x;
  x = y;
  y = temp;
  // print de nbPassagers affiche 20, 10
}
```

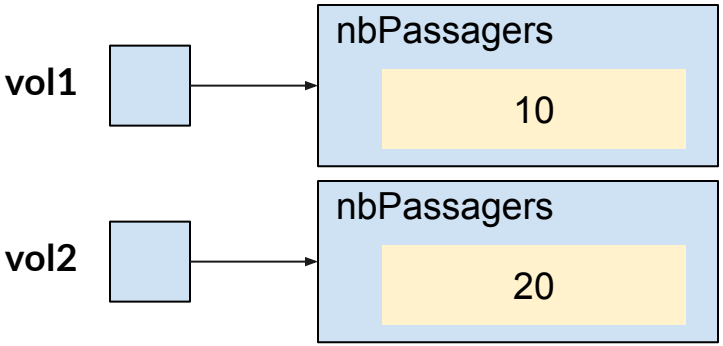
Exemple avec types références




```
Vol vol1 = new Vol(10);
Vol vol2 = new Vol(20);
// print de nbPassagers affiche 10, 20

echanger(vol1, vol2);
```

```
void echanger(Vol x, Vol y) {
  Vol temp = x;
  x = y;
  y = temp;
  // print de nbPassagers affiche 20, 10
}
```

Exemple avec types références



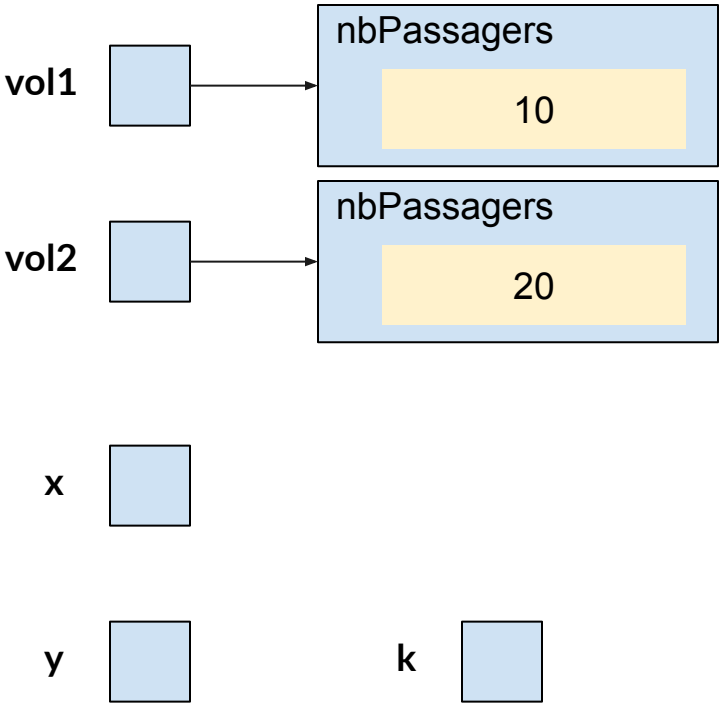

```
Vol vol1 = new Vol(10);
Vol vol2 = new Vol(20);
// print de nbPassagers affiche 10, 20

echanger(vol1, vol2);
// print de nbPassagers affiche toujours 10,20
```

```
void echanger(Vol x, Vol y) {
  Vol temp = x;
  x = y;
  y = temp;
  // print de nbPassagers affiche 20, 10
}
```

Immuabilité des paramètres

- Lors de l'appel d'une méthode, chaque argument est passé en faisant une copie de la valeur (passage par valeur, *by-value*)
- Le paramètre qui reçoit la valeur agit donc comme une variable locale à la méthode : tout changement de valeur sur le paramètre n'affectera pas l'argument d'origine
- **En revanche, si l'argument est un type-référence, les éventuels changements effectués *sur l'objet pointé* resteront effectifs à la sortie de la méthode**

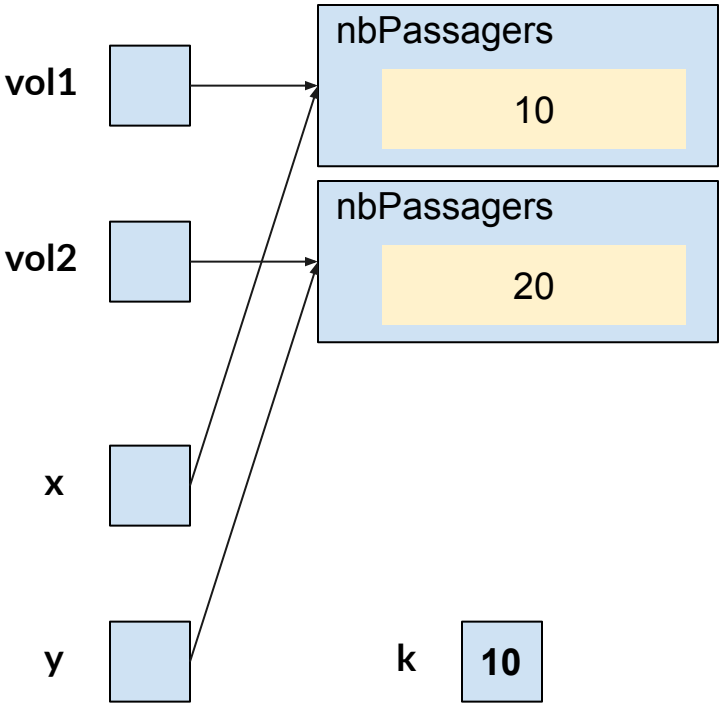
Exemple 2 avec types références




```
Vol vol1 = new Vol(10);
Vol vol2 = new Vol(20);
// print de nbPassagers affiche 10, 20

echangerNbPassagers(vol1, vol2);
```

```
void echangerNbPassagers(Vol x, Vol y) {
  int temp = x.getNbPassagers();
  x.setNbPassagers(y);
  y.setNbPassagers(temp);

}
```

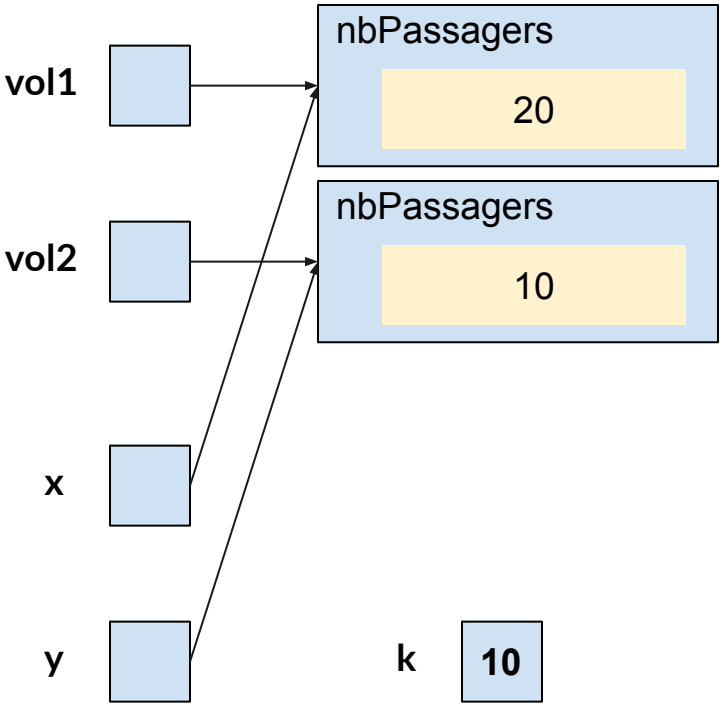
Exemple 2 avec types références




```
Vol vol1 = new Vol(10);
Vol vol2 = new Vol(20);
// print de nbPassagers affiche 10, 20

echangerNbPassagers(vol1, vol2);
```

```
void echangerNbPassagers(Vol x, Vol y) {
  int temp = x.getNbPassagers();
  x.setNbPassagers(y);
  y.setNbPassagers(temp);

}
```

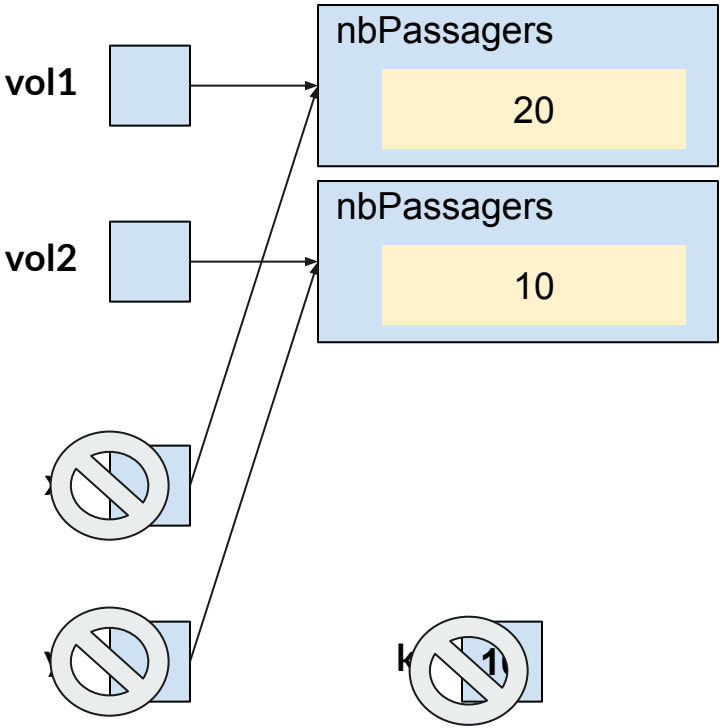
Exemple 2 avec types références




```
Vol vol1 = new Vol(10);
Vol vol2 = new Vol(20);
// print de nbPassagers affiche 10, 20

echangerNbPassagers(vol1, vol2);
```

```
void echangerNbPassagers(Vol x, Vol y) {
  int temp = x.getNbPassagers();
  x.setNbPassagers(y.getNbPassagers());
  y.setNbPassagers(temp);
  // print de nbPassagers affiche 20, 10
}
```

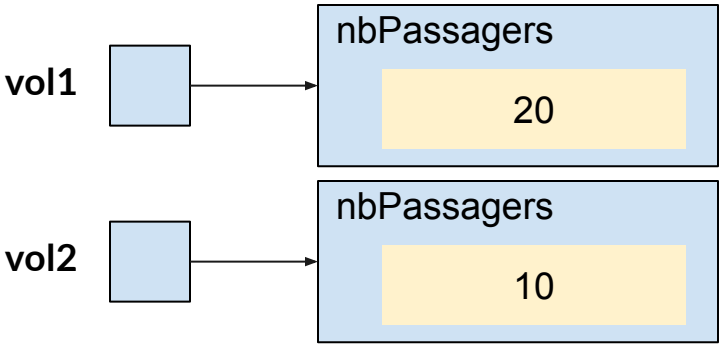
Exemple 2 avec types références




```
Vol vol1 = new Vol(10);
Vol vol2 = new Vol(20);
// print de nbPassagers affiche 10, 20

echangerNbPassagers(vol1, vol2);
```

```
void echangerNbPassagers(Vol x, Vol y) {
  int temp = x.getNbPassagers();
  x.setNbPassagers(y.getNbPassagers());
  y.setNbPassagers(temp);
  // print de nbPassagers affiche 20, 10
}
```

Exemple 2 avec types références




```
Vol vol1 = new Vol(10);
Vol vol2 = new Vol(20);
// print de nbPassagers affiche 10, 20

echangerNbPassagers(vol1, vol2);
// print de nbPassagers affiche 20, 10
```

```
void echangerNbPassagers(Vol x, Vol y) {
  int temp = x.getNbPassagers();
  x.setNbPassagers(y.getNbPassagers());
  y.setNbPassagers(temp);
  // print de nbPassagers affiche 20, 10
}
```

Surchage de méthode (*overloading*)

- Une classe peut avoir **plusieurs versions** de constructeurs et de méthodes (même nom)
- Ce procédé (**surcharge**, ***overloading***) a été rencontré lorsque nous avons défini plusieurs constructeurs pour une classe
- Pour distinguer les méthodes, le compilateur vérifie la **signature** des méthodes :
  - nom de la méthode
  - nombre de paramètres
  - type de chaque paramètre
- Les règles de sélection (compatibilité de types) sont en général intuitives (ça fait ce que vous attendez), mais pour tout savoir : bit.ly/javamethodselection

```
class Vol {
  int nbPlaces = 150;
  int nbPassagers;
  int nbBagagesEnregistres;
  int maxBagagesAMain = nbPlaces * 2;
  int nbBagagesAMain;

  // Constructeurs et autres membres…

  public void ajouterUnPassager() {
    if (placeDisponible())
      nbPassagers++;
    else
      GererTropNombreux();
  }

  private boolean placeDisponible() {
    return nbPassagers < nbPlaces;
  }

  private boolean placePourBagagesAMain(int nb) {
    return nbBagagesAMain + nb <= maxBagagesAMain;
  }
}
```

```
class Vol {
  // autres membres…

  public void ajouterUnPassager() {
    if (placeDisponible())
      nbPassagers++;
    else
      GererTropNombreux();
  }

}
```

```
class Vol {
  // autres membres…

  public void ajouterUnPassager() {
    if (placeDisponible())
      nbPassagers++;
    else
      GererTropNombreux();
  }

  public void ajouterUnPassager(int nbBagagesEnregistres) {
    if (placeDisponible) {
      ajouterUnPassager();
      this.nbBagagesEnregistres += nbBagagesEnregistres;
    }
  }
}
```

```
class Vol {
  // autres membres…

  public void ajouterUnPassager() {
    if (placeDisponible())
      nbPassagers++;
    else
      GererTropNombreux();
  }

  public void ajouterUnPassager(int nbBagagesEnregistres) {
    if (placeDisponible) {
      ajouterUnPassager();
      this.nbBagagesEnregistres += nbBagagesEnregistres;
    }
  }

  public void ajouterUnPassager(Passager p) {
    ajouterUnPassager(p.getNbBagagesEnregistres());
  }
}
```

```
class Vol {
  // autres membres…

  public void ajouterUnPassager() { … }
  public void ajouterUnPassager(int nbBagagesEnregistres) { … }
  public void ajouterUnPassager(Passager p) { … }

  public void ajouterUnPassager(int nbBagagesEnregistres, int nbBagagesAMain) {
    if (placeDisponible() && placePourBagagesAMain(nbBagagesAMain)) {
      ajouterUnPassager(nbBagagesEnregistres);
      this.nbBagagesAMain += nbBagagesAMain;
    }
  }
}
```

```
class Vol {
  // autres membres…

  public void ajouterUnPassager() { … }
  public void ajouterUnPassager(int nbBagagesEnregistres) { … }
  public void ajouterUnPassager(Passager p) { … }

  public void ajouterUnPassager(int nbBagagesEnregistres, int nbBagagesAMain)
{
    if (placeDisponible() && placePourBagagesAMain(nbBagagesAMain)) {
      ajouterUnPassager(nbBagagesEnregistres);
      this.nbBagagesAMain += nbBagagesAMain;
    }
  }

  public void ajouterUnPassager(Passager p, int nbBagagesAMain) {
    ajouterUnPassager(p.getNbBagagesEnregistres(), nbBagagesAMain);
  }
}
```

```
class Vol {
  // autres membres…

  public void ajouterUnPassager() { … }
  public void ajouterUnPassager(int nbBagagesEnregistres) { … }
  public void ajouterUnPassager(Passager p) { … }
  public void ajouterUnPassager(int nbBagagesEnregistres,int nbBagagesAMain){}
  public void ajouterUnPassager(Passager p, int nbBagagesAMain) { … }
}
```

```
Vol v = new Vol();
Passager p1 = new Passager(0, 1); // (nbBagagesAMain, nbBagagesEnregistres)
Passager p2 = new Passager(0, 2);

v.ajouterUnPassager();
v.ajouterUnPassager(2);
v.ajouterUnPassager(p1);

short troisBagages = 3;
v.ajouterUnPassager(troisBagages, 2);
v.ajouterUnPassager(p2, 1);
v.ajouterUnPassager(4L); // Erreur compilation, conversion long->int interdite
```

Nombre de paramètres variable

```
Vol v = new Vol();
Passager tom = new
               Passager(0, 1);
Passager lea = new
               Passager(0, 2);
Passager bob = new
               Passager(0, 0);

v.ajouterPassagers(
 new Passager[] {tom, lea,
                 bob});
```

```
class Vol {
  public void ajouterPassagers(Passager[] liste)
{
    if (placeDisponiblePour(liste.length)) {
      nbPassagers += liste.length;
      for (Passager p : liste) {
        nbBagagesEnregistres
          += p.getNbBagagesEnregistres();
      }
    }
    else
      GererTropNombreux();
  }

  private boolean placeDisponiblePour(int nb) {
    return nbPassagers + nb <= nbPlaces;
  }
}
```

Nombre de paramètres variable

- Une méthode peut être déclarée comme acceptant un nombre variable de paramètres
- Placez une ellipse (trois points) après le type du paramètre
- Seulement le dernier paramètre de la méthode
- La méthode reçoit les arguments en tableau

```
class Vol {
  public void ajouterPassagers( Passager... liste){
    if (placeDisponiblePour(liste.length)) {
      nbPassagers += liste.length;
      for (Passager p : liste) {
        nbBagagesEnregistres
          += p.getNbBagagesEnregistres();
      }
    }
    else
      GererTropNombreux();
  }

  private boolean placeDisponiblePour(int nb) {
    return nbPassagers + nb <= nbPlaces;
  }
}
```

Nombre de paramètres variable

```
Vol v = new Vol();
Passager tom = new
               Passager(0, 1);
Passager lea = new
               Passager(0, 2);
Passager bob = new
               Passager(0, 0);

v.ajouterPassagers(tom, lea,
                   bob);
```

```
class Vol {
  public void ajouterPassagers( Passager... liste){
    if (placeDisponiblePour(liste.length)) {
      nbPassagers += liste.length;
      for (Passager p : liste) {
        nbBagagesEnregistres
          += p.getNbBagagesEnregistres();
      }
    }
    else
      GererTropNombreux();
  }

  private boolean placeDisponiblePour(int nb) {
    return nbPassagers + nb <= nbPlaces;
  }
}
```

Résumé

- Les **paramètres sont immuables**
  - changements sur les **paramètres** sont **invisibles une fois sorti**
  - changements sur les **membres pointés** sont **persistants**
- Plusieurs « versions » possibles pour un même nom de méthode (***overloading***)
  - Condition : **signature unique** pour chacune
  - signature = nom + nombre paramètres + types paramètres
- **Nombre variable de paramètres**
  - *syntactic sugar* permettant d'énumérer les arguments directement lors de l'appel
  - ellipse (...) dans la déclaration de méthode sur le dernier paramètre
  - en interne : c'est un simple tableau