Programmation Par Objets

Généricité

C'est quoi?

- Capacité d'écrire du code où certains constituants (généralement des types, mais aussi des fonctions) ne sont pas fixés
- Par fixation de ces constituants, on obtient alors autant de versions de ce code
- Exemples:
  - Des algorithmes indépendants des types de données manipulées
    - Algorithmes de tri sur des tableaux d'éléments de n'importe quel type muni d'une relation d'ordre
    - Algorithmes de graphes ou de recherche opérationnelle, paramétrés par des fonctions de distance, de coût, …
  - Des SD offrant des stratégies d'implantation variées, indépendamment des types d'éléments
  - Et plus généralement, des applications offrant des mécanismes indépendants des types d'entités manipulées
    - Frameworks architecturaux (J2EE)
    - Frameworks applicatifs
      - E-commerce
      - Gestion de ressources
      - …

Objectifs

- Factorisation et montée en abstraction
- Réutilisation
- Economie de code
  - et productivité
- Vérification au plus tôt
  - Sureté
    - vérification une seule fois sur les mécanismes offerts (algorithmes, structures)
  - Idéalement à la compilation
    - vérification statique

Deux formes principales de généricité

I. Généricité ou polymorphisme d'***inclusion***

- Propre à la PPO
- Dûe à la notion de sous-typage ou d'inclusion qu'elle induit sur ses hiérarchies de classes et d'interfaces
- Principe: **substitution**
  - Partout où on s'attend à avoir un objet d'un type T, un objet d'un sous-type de T convient
- D'où la programmation d'hiérarchies de classes
  - Où les surclasses sont génériques, parce qu'applicables à toute spécialisation (sous-classe) de this
    - Cf. méthodes génériques par this-message
  - Permettant la généricité (et l'évolution) des applications
    - Une application à un niveau de la hiérarchie s'applique aux sous-niveaux
  - Exemples :
    - Traitements de Banque sur hiérarchie de classes de Compte
    - Dessin sur hiérarchie de classes de Figure
    - Manipulations de Circuit sur hiérarchie de classes de Porte

Deux formes principales de généricité

II. Généricité ou polymorphisme ***paramétrique***

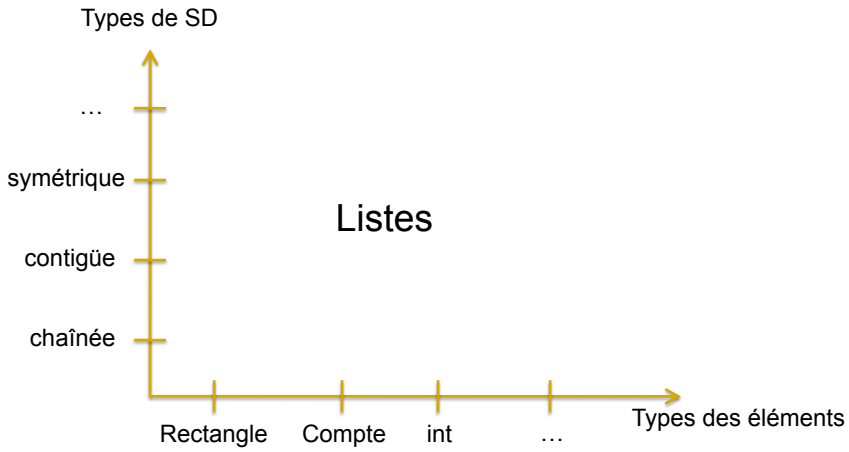
- Principe: **paramétrage** du code par des types ou des fonctions
- Pas propre à la PPO
  - formes primitives: pointeurs de fonctions en C
  - formes abouties en ADA: paramètres de type, de procédures et fonctions, d'opérateurs
- Mais intégré dans la plupart des langages à objets
  - avec contrôle statique de types (compilation)
  - génériques en Java ou C#, templates C++

Démarche du cours

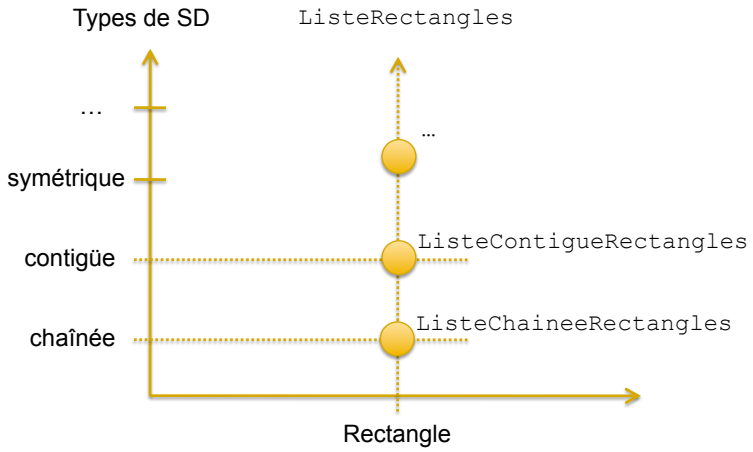
- Etude de cas
  - 2 axes de factorisation de code
- polymorphisme d'inclusion: connu
- mais ne résout pas tout…
- besoins de généricité paramétrique
- en Java

  mais les principes sont généraux

Etude de cas: 2 axes de factorisation



I. Factorisation verticale: différents types de SD sur un même type d'éléments



```
public class ListeContigueRectangles {
    int max_size, nbElements;
    Rectangle[] tab;

    public int size() {return nbElements;}
    public void add(Rectangle x) throws ListePleineException {
        if (this.nbElements == this.max_size) {
            throw new ListePleineException();
        } else {
            this.tab[this.nbElements] = x;
            this.nbElements++;
} … }

public class ListeChaineeRectangles {
    int max_size, int nbElements;
    Cellule first;

    public int size() {return nbElements;}
    public void add(Rectangle x) throws ListePleineException {
        if (this.nbElements == this.max_size) {
            throw new ListePleineException();
        } else {
            Cellule tmp = new Cellule(x, first); first = tmp;
            this.nbElements++;
} … }
```



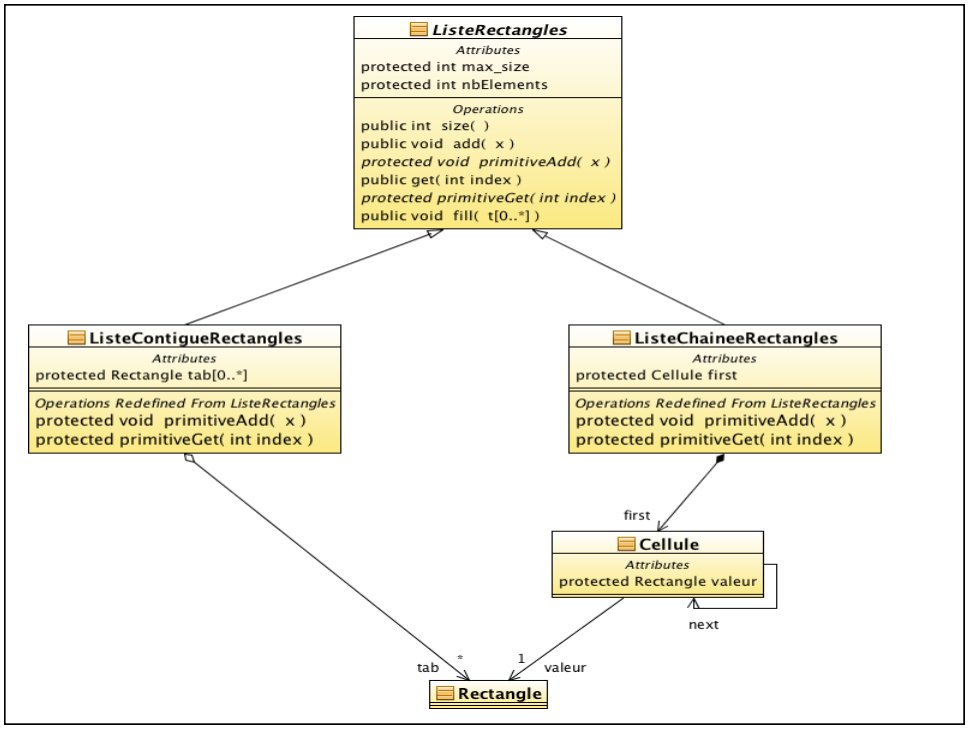
```
public class ListeContigueRectangles {
    int max_size, nbElements;
    Rectangle[] tab;

    public int size() {return nbElements;}
    public void add(Rectangle x) throws ListePleineException {
        if (this.nbElements == this.max_size) {
            throw new ListePleineException();
        } else {
            this.tab[this.nbElements] = x;
            this.nbElements++;
} … }

public class ListeChaineeRectangles {
    int max_size, int nbElements;
    Cellule first;

    public int size() {return nbElements;}
    public void add(Rectangle x) throws ListePleineException {
        if (this.nbElements == this.max_size) {
            throw new ListePleineException();
        } else {
            Cellule tmp = new Cellule(x, first); first = tmp;
            this.nbElements++;
} … }
```

Solution:
Hiérarchie de classes
(inclusion)




```
public abstract class ListeRectangles {
    int max_size;
    int nbElements;

    // implementation : see subclasses

    public int size() {return nbElements;}

    public void add(Rectangle x) throws ListePleineException {
        if (this.nbElements == this.max_size) {
            throw new ListePleineException();
        } else {
            this.primitiveAdd(x); // generique (inclusion)
            this.nbElements++;
        }
    }
    protected abstract void primitiveAdd(Rectangle x);

    public void fill(Rectangle[] t) throws ListePleineException {
        for (Rectangle x : t) {
            this.add(x);
        }
    }
    // idem pour: Rectangle get(i)…
}
```

```
public class ListeContigueRectangles extends ListeRectangles {
    Rectangle[] tab; //implementation contigue

    protected void primitiveAdd(Rectangle x) {
        this.tab[this.nbElements] = x;
    }
    protected Rectangle primitiveGet(int index) {
        return tab[index];
    }
}
public class ListeChaineeRectangles extends ListeRectangles {
    Cellule first; //implementation chainee

    protected void primitiveAdd(Rectangle x) {
        Cellule tmp = new Cellule(x, first);
        first = tmp;
    }
    protected Rectangle primitiveGet(int index) {
        Cellule cell = first;
        for (int i = 0; i < index; i++) {
            cell = cell.getNext();
        }
        return cell.getValeur();
    }
}
```

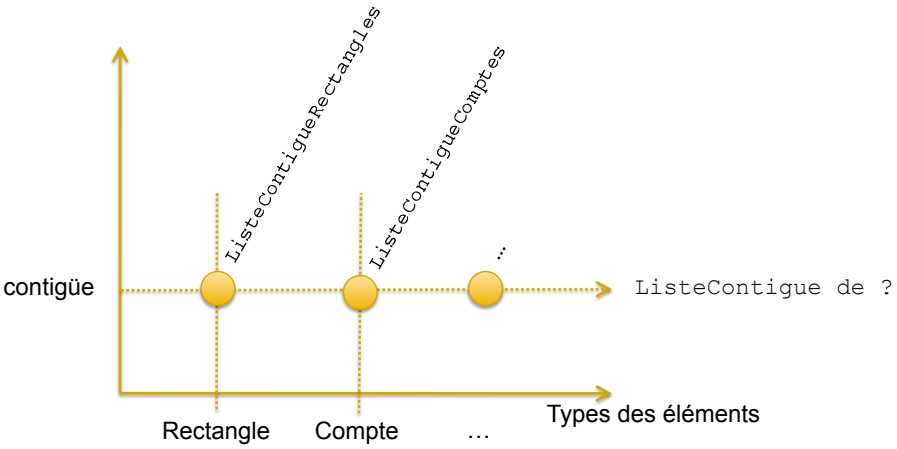

Généricité à l'utilisation

```
// main ou plus generalement classe utilisatrice
        // polymorphisme d'inclusion:
        ListeRectangles lContigue = new ListeContigueRectangles(5);
        ListeRectangles lChainee = new ListeChaineeRectangles(5);

        initListe(lContigue); printListe(lContigue);
        initListe(lChainee); printListe(lChainee);

void initListe(ListeRectangles l) {// generique (inclusion)
        try {
            l.add(new Rectangle(0.0, 0.0, 10.0, 20.0));
            l.add(new Rectangle(50.0, 50.0, 150.0, 250.0));
            l.add(new Rectangle(100.0, 100.0, 1100.0, 2100));
        } catch (ListePleineException ex) {
            System.out.println("full!");
}}
void printListe(ListeRectangles l) { // generique (inclusion)
        for (int i = 0; i < l.size(); i++) {
            System.out.println("i=" + i + " " + l.get(i));
}}
```

II. Factorisation horizontale: un même type de SD sur différents types d'éléments




```
public class ListeRectangles {
    int max_size, nbElements;
    Rectangle[] tab;

    public int size() {return nbElements;}
    public void add(Rectangle x) throws ListePleineException {
        if (this.nbElements == this.max_size) {
            throw new ListePleineException();
        } else {
            this.tab[this.nbElements] = x;
            this.nbElements++;
} … }

public class ListeComptes {

    int max_size, nbElements;
    Compte[] tab;

    public int size() {return nbElements;}

    public void add(Compte x) throws ListePleineException {
        if (this.nbElements == this.max_size) {
            throw new ListePleineException();
        } else {
            this.tab[this.nbElements] = x;
            this.nbElements++;
} … }
```

```
public class ListeRectangles {
    int max_size, nbElements;
    Rectangle[] tab;

    public int size() {return nbElements;}
    public void add(Rectangle x) throws ListePleineException {
        if (this.nbElements == this.max_size) {
            throw new ListePleineException();
        } else {
            this.tab[this.nbElements] = x;
            this.nbElements++;
} … }

public class ListeComptes {

    int max_size, nbElements;
    Compte[] tab;

    public int size() {return nbElements;}

    public void add(Compte x) throws ListePleineException {
        if (this.nbElements == this.max_size) {
            throw new ListePleineException();
        } else {
            this.tab[this.nbElements] = x;
            this.nbElements++;
} … }
```

Idée:
Surtype commun: Object
Et profiter du polymorphisme d'inclusion pour ranger des Rectangles ou des Comptes ou …


```
public class ListeContigue {
  int max_size, nbElements;
  Object[] tab;

  public int size() {return nbElements;}

  public void add(Object x) throws ListePleineException {
      if (this.nbElements == this.max_size) {
          throw new ListePleineException();
      } else {
          this.tab[this.nbElements] = x;
          this.nbElements++;
  }}

  public Object get(int index) throws IndexOutOfBoundsException {
      Object tmp = null;
      if (index < 0 || index >= this.nbElements) {
          throw new IndexOutOfBoundsException();
      } else {
          tmp = this.tab[index];
      }
      return tmp;
  }
…}
```

Généricité à l'utilisation

```
ListeContigue lrects = new ListeContigue(5);
ListeContigue lcomptes = new ListeContigue(5);

initRectangles(lrects); printListe(lrects);
initComptes(lcomptes); printListe(lcomptes);

void initRectangles(ListeContigue l) { // non generique : depend du type
       try {
           l.add(new Rectangle(0.0, 0.0, 10.0, 20.0)); // … OK Object's
      } catch (ListePleineException ex) {System.out.println("full! »); }
}
void initComptes(ListeContigue l) {// non generique : depend du type
       try {
           l.add(new Compte(1000, 500.0)); // … OK Object's
       } catch (ListePleineException ex) {System.out.println("full!");}
}
void printListe(ListeContigue l) { // generique sur Object (inclusion)
       for (int i = 0; i < l.size(); i++) {
           System.out.println("i=" + i + " " + l.get(i)); // toString() suffit
       }
}
```


Mais…

```
// somme des surfaces de la liste de Rectangles…
Rectangle r;
double somme = 0.0;
for (int i=0; i<lrects.size(); i++) {
  r = lrects.get(i);
  somme += r.surface();
}
```

Erreur de compilation:
Incompatible types
found : Object
required: Rectangle

- Solution: cast « forcé »

```
Rectangle r;
double somme = 0.0;
for (int i=0; i<lrects.size(); i++) {
  r = (Rectangle)lrects.get(i);
  somme += r.surface();
}
```

OOF!
Le compilateur accepte
Et l'exécution marche

Mais…

```
initRectangles(lrects);
lrects.add(new Compte(100.0, 50.0));
printListe(lrects);

Rectangle r;
double somme = 0.0;
for (int i=0; i<lrects.size(); i++) {
  r = (Rectangle)lrects.get(i);
  somme += r.surface();
}
```

Plus gênant!
Le compilateur accepte aussi…

- Pour le compilateur
  - Seule contrainte vérifiable : `lrects` (tout comme `lcomptes`) doivent contenir des `Objects`
  - `Rectangle` et `Compte` sont des `Objects` (polymorphisme d'inclusion)
- Et à l'exécution…

Mais…

```
initRectangles(lrects);
lrects.add(new Compte(100.0, 50.0));
printListe(lrects);

Rectangle r;
double somme = 0.0;
for (int i=0; i<lrects.size(); i++) {
  r = (Rectangle)lrects.get(i);
  somme += r.surface();
}
```

Plus gênant!
Le compilateur accepte aussi…

- Et à l'exécution…
  - **`printListe` OK (`Object` suffit)**

```
i=0 Rectangle
i=1 Rectangle
i=2 Compte
```

  - **mais surfaces… boom!**
    Type dynamique (Compte) incompatible avec le type statique (Rectangle)

```
BOOM!
Exception
ClassCastException:
Compte cannot be cast
to Rectangle
```

Bilan

- `ListeContigue` (d'`Object`) est « trop générique »
- On a bien des **objets listes:** listes de rectangles, de comptes (et de n'importe quoi…)
  - en profitant du polymorphisme d'inclusion sur les éléments tous compatibles `Object`
- Mais pas de **types de Liste**: Liste de Rectangle, Liste de Compte
  - Avec le contrôle statique de type des éléments correspondants
  - Sauf casts systématiques et coûteux (vérification dynamique à l'exécution)
- Solution ?
  - programmer toutes ces classes « à la main »
  - fastidieux, risque d'erreur (duplication de code) et peu lisible
  - alors que leur code ne diffère que par le type des éléments

C'est l'objectif de la **généricité (ou polymorphisme) paramétrique**
Paramétrer du code (classes, méthodes, …) par des types
Garantir les vérifications correspondantes dès la compilation (fiabilité)

Généricité paramétrique

```
public class ListeContigue<E> {
    int max_size, nbElements;
    E[] tab;

    public void add(E x) throws ListePleineException {
        if (this.nbElements == this.max_size) {
            throw new ListePleineException();
        } else {
            this.tab[this.nbElements] = x; this.nbElements++;
    }}
    public E get(int index) throws IndexOutOfBoundsException {
        E tmp = null;
        if (index < 0 || index >= this.nbElements) {
            throw new IndexOutOfBoundsException();
        } else {
            tmp = this.tab[index];
        }
        return tmp;
    } …}
```

- **`ListeContigue<E>` : classe générique**
- **`E` : paramètre formel générique de type**

Instanciation de classe générique

- Par fixation des paramètres génériques de type par des types effectifs
- On obtient autant de versions de la classe correctement typées

```
ListeContigue<Rectangle>
ListeContigue<Compte>
```

Rectangle, Compte: paramètres effectifs génériques de types

- Utilisable comme toute classe:

```
ListeContigue<Rectangle> lrects;
ListeContigue<Compte> lcomptes;
```

- Instanciable (sauf si abstract, cf. plus loin)

```
lrects = new ListeContigue<Rectangle>(5);
```

- Attention, ne pas confondre
  - Instances de classes génériques : `ListeContigue<Rectangle>` qui sont des classes
  - et instances de celles-ci: `new ListeContigue<Rectangle>(10)` qui sont des objets

Utilisation

```
// main ou plus generalement classe utilisatrice
ListeContigue<Rectangle> lrects
  = new ListeContigue<Rectangle>(5);
ListeContigue<Compte> lcomptes
  = new ListeContigue<Compte>(5);
```

Rectangle, Compte: types génériques effectifs

- On a gagné: vérification statique

```
Rectangle r;
double somme = 0.0;
for (int i=0; i<lrects.size(); i++) {
  r = lrects.get(i);
  somme += r.surface();
}
```

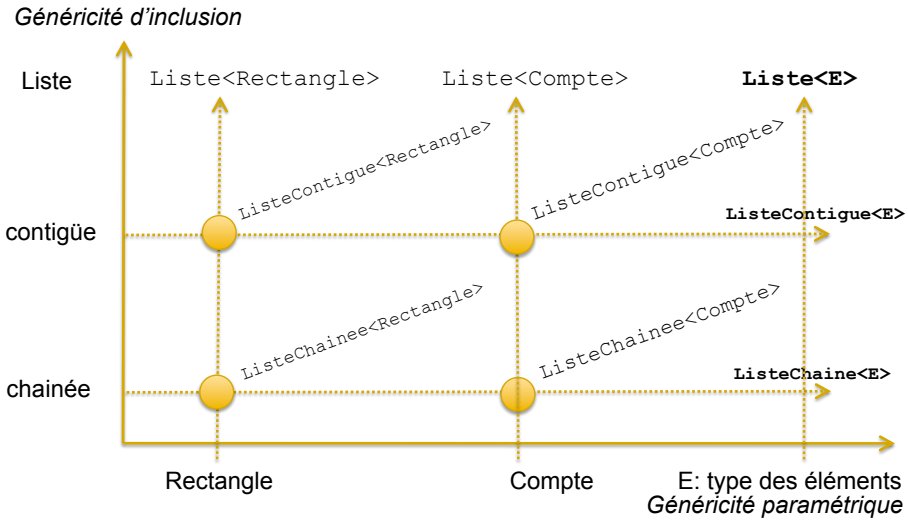
Compilateur : OK
plus de cast nécessaire

- Et le compilateur n'accepte plus :

```
lrects.add(new Compte(100.0, 50.0));
```

Erreur de compilation: add(Rectangle) cannot be applied to Compte

La synthèse: hiérarchie de classes génériques




```
public abstract class Liste<E> {
    int max_size;
    int nbElements;
    // implementation : see subclasses

    public int size() {return nbElements;}

    public void add(E x) throws ListePleineException {
        if (this.nbElements == this.max_size) {
            throw new ListePleineException();
        } else {
            this.primitiveAdd(x); // generique (inclusion)
            nbElements++;
        }}
    protected abstract void primitiveAdd(E x);

    public void fill(E[] t) throws ListePleineException {
        for (E x : t) {
            this.add(x);
        }
    }
    // idem pour: E get(i)…
}
```

Classe générique abstraite: ses instances le seront

```
public class ListeContigue<E> extends Liste<E> {
    E[] tab; //implementation contigue
    protected void primitiveAdd(E x) {
        this.tab[this.nbElements] = x;
    }
    protected E primitiveGet(int index) {
        return tab[index];
    }
}

public class ListeChainee<E> extends Liste<E> {
    Cellule<E> first; //implementation chainee
    protected void primitiveAdd(E x) {
        Cellule<E> tmp = new Cellule<E>(x, first);
        first = tmp;
    }
    protected E primitiveGet(int index) {
        Cellule<E> cell = first;
        for (int i = 0; i < index; i++) {
            cell = cell.getNext();
        }
        return cell.getValeur();
    }
```


Et toujours: généricité d'inclusion à l'utilisation

classe abstraite

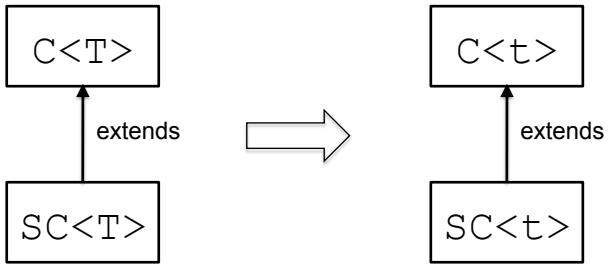
```
// polymorphisme d'inclusion:
    Liste<Rectangle> lContigueRectangles = new ListeContigue<Rectangle>(5);
    Liste<Rectangle> lChaineeRectangles = new ListeChainee<Rectangle>(5);

    initListe(lContigueRectangles ); printListe(lContigueRectangles );
    initListe(lChaineeRectangles); printListe(lChaineeRectangles);
}
 void initListe(Liste<Rectangle> l) {// generique (inclusion)
    try {
        l.add(new Rectangle(0.0, 0.0, 10.0, 20.0));
        l.add(new Rectangle(50.0, 50.0, 150.0, 250.0));
        l.add(new Rectangle(100.0, 100.0, 1100.0, 2100));
    } catch (ListePleineException ex) {
        System.out.println("full!");
}}
  void printListe(Liste<Rectangle> l) { // generique (inclusion)
    for (int i = 0; i < l.size(); i++) {
        System.out.println("i=" + i + " " + l.get(i));
}}
```

Hiérarchie de classes génériques et sous-typage

Règle (sous-typage de classes génériques):

- *si* `SC<T>` *extends* `C<T>`
- *alors* `SC<t>` *extends* `C<t>`
  pour toute type `t`



Paramètres génériques et sous-typage

- Comment factoriser (rendre générique):

```
void printRectangles(Liste<Rectangle> l) {
 for (int i = 0; i < l.size(); i++) {
   System.out.println("i=" + i + " " + l.get(i));
 }}
void printComptes(Liste<Compte> l) {
// idem
}
```

Idée:
Object:toString() suffit

- Solution?

```
void printListe(Liste<Object> l) {
  for (int i = 0; i < l.size(); i++) {
   System.out.println("i=" + i + " " + l.get(i));
 }}
```

Compilation: OK

Paramètres génériques et sous-typage

- Mais à l'utilisation, ça ne marche pas…

```
Liste<Compte> lcomptes;
Liste<Rectangle> lrectangles;

printListe(lcomptes);
printListe(lrectangles)
```

Erreur de compilation:
printListe(Liste<Object>) cannot be applied to Liste<Compte>)

Idem pour lrectangles

- Pourquoi `Liste<Compte>` n'est pas sous-type de `Liste<Object>` ?

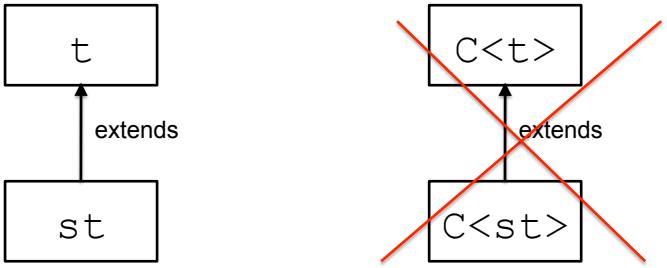
Paramètres génériques et sous-typage

- Raisonnement par l'absurde:
  - Supposons que:
    `Liste<Compte>` soit sous-type de `Liste<Object>`
  - Alors:
    - `Liste<Object> lobjects = lcomptes;` serait valide
    - la méthode `add(Object)` de `Liste<Object>` serait applicable
    - et donc on pourrait ranger n'importe quel objet dans la liste de comptes (`lcomptes`) « à travers la variable » `lobjects`
    - par exemple un rectangle!
      `lobjects.add(new Rectangle(…));`
  - CQFD

Paramètres génériques et sous-typage

Règle (sous-typage de paramètre générique):

- *Soit* `C<T>` *une classe générique*
- `st` *sous-type de* `t`
- `C<st>` *n'est pas sous-type de* `C<t>`



Solution : paramètres jokers (« wildcards »)

```
void printListe(Liste<?> l) {
  for (int i = 0; i < l.size(); i++) {
   System.out.println("i=" + i + " " + l.get(i));
}}
```

Règle (joker):

- *Soit* `C<T>` *une classe générique*
- `C<?>` *(et non* `C<Object>`*) est le sur-type commun à toutes les classes* `C<t>`

Remarque: on peut aussi écrire:
`<T> void printListe(Liste<T> l) {…}`
mais le nommage de `T` est inutile dans le code de `printListe`

jokers : limitations

- Rappel:

`Liste<Compte>` n'est pas sous-type de `Liste<Object>`

```
Liste<Object> lobjects = lcomptes;
lobjects.add(new Rectangle(…));
```

- joker:

`Liste<Compte>` est sous-type de `Liste<?>`

```
Liste<?> l = lcomptes; // OK
```

- Mais tout n'est pas permis pour autant sur `l`!

```
l.add(new Rectangle(…)); // toujours mais aussi…
l.add(new Compte(…)); // bizare!?
```

- En effet `l` peut contenir des listes de n'importe quoi:

```
l = lrectangles; // OK
```

De manière générale le compilateur refuse toutes les opérations de modification sur `C<?>` (type « inconnu »).

Généricité contrainte (ou bornée) inférieurement

- Solution:

```
Liste<? super Compte> l = lcomptes;
```

  - Et on peut…

```
l.add(new Compte(…)); // OK cette fois
```

  - Ainsi que… (toute liste d'Object peut contenir des comptes)

```
l = lobjects; // avec: Liste<Object> lobjects
l.add(new Compte(…));
```

  - Mais pas (plus):

```
l.add(new Rectangle(…));
l = lrectangles;
```

Erreur de compilation:
`Rectangle` incompatible avec
`<? super Compte>`

Généricité contrainte (ou bornée) inférieurement

Règle (borne inférieure):

- *Soit* `C<T>` *une classe générique*
- `C<? super t>` *est compatible avec* `C<t'>` *, pour tout* `t'` *sur-type de* `t`

- Applications typiques: modifications génériques
  - ajouts/suppressions
  - copies
  - initialisation

Généricité contrainte (ou bornée) inférieurement

Exemple

- en supposant que `Rectangle` et `Cercle` soient sous-classes de:

```
public abstract class Figure { … }
```

- Initialisation par un tableau de rectangles

```
void initRects(Liste<? super Rectangle> dest, Rectangle[] src){
   for (Rectangle x : src) { dest.add(x);}
}
```

- applicable à tout type de listes pouvant contenir des rectangles:
  `Liste<Rectangle>`, `Liste<Figure>`, `Liste<Object>`
- mais pas à:
  `Liste<Cercle>`, `Liste<Compte>`

Généricité contrainte (ou bornée) inférieurement

Généralisation de « Rectangle » à tout `T`

- initialisation de tout type de listes pouvant contenir des `T` par un tableau de `T`. Nécessité de nommer `T` pour l'identifier:

```
<T> void init(Liste<? super T> dest, T[] src) {
        for (T x : src) {dest.add(x);}
}
```

- Applicable à :
  - Comme ci-dessus

```
init(lrectangles, tabRects); init(lfigures, tabRects);
```

  - Et aussi

```
init(lcercles, tabCercles); init(lfigures, tabCercles);
init(lcomptes, tabComptes);
init(lobjects, tabRects); init(lobjects, tabComptes); //…
```

  - Mais pas

```
init(lcercles, tabRects); init(lcomptes, tabRects); //…
```


Généricité bornée **supérieurement**

- En supposant que `Rectangle` et `Cercle` soient sous-classes de:

```
public abstract class Figure {
    public abstract double surface();
}
```

- Comment factoriser (rendre générique) ?

```
double surfacesRectangles(Liste<Rectangle> l) {
        double somme = 0.0; // des surfaces
        for (int i=0; i<l.size(); i++) {
            somme += l.get(i).surface();
        }
        return somme;
}
double surfacesCercles(Liste<Cercle> l) {
      // meme code
}
```

Généricité contrainte (ou bornée) supérieurement

- Idée? `Figure` est sur-type de `Rectangle, Cercle, …` :

```
double surfaces(Liste<Figure> l) {…}
```

  - ☺ ça compile
  - ☹ mais ce n'est pas utilisable sur des listes de rectangles, de cercles
  - en effet (règle de sous-typage de paramètres génériques): `Liste<Rectangle>`, `Liste<Cercle>`, … ne sont sous-types de `Liste<Figure>`

- Autre idée? `Liste<?>` est sur-type de `Liste<Rectangle>`, `Liste<Cercle>, …`:

```
double surfaces(Liste<?> l) {…}
```

  - ☹ ça ne compile pas…
  - normal! Le type est inconnu = joker `?` (ou `extends Object`) donc pas de méthode `surface()` garantie

Généricité contrainte (ou bornée) supérieurement

- Solution: généricité bornée **supérieurement**

```
double surfaces(Liste<? extends Figure> l) {// meme code }
```

  - Applicable à tout type de listes de sous-types de `Figure`: `Liste<Figure>, Liste<Rectangle>, Liste<Cercle>`
  - Mais pas `Liste<Object>, Liste<Compte>`

Règle (borne supérieure):

- *Soit* `C<T>` *une classe générique*
- `C<? extends t>` *est compatible avec* `C<t'>` *, pour tout* `t'` *sous-type de* `t`

Remarque:

`C<?>` ⇔ `C<? extends Object>`

A savoir également

- Paramètres génériques multiples
  - de classe générique

```
class HashMap<K,V>
  // K type de clef, V type de valeur
```

  - de méthode

```
<K,V> void ajoutCouple(HashMap<K,V> h, K key, V val) {
       h.put(key,val);
}
```

- Il est possible de créer des classes (non génériques) qui héritent de classes génériques « instanciées »

```
class Contacts extends HashMap<String,Personne> {…}
```


Pour conclure

- En Java la généricité a été introduite tardivement : version 5.0 en 2005, alors que Java est né en 1995
- La génération de code procède par « effacement de type »
  - Effacement de toutes les informations entre < >

    `class Liste<E>` => `class Liste`
  - `Liste` est appelé « type brut » (« raw type »)
  - Les types formels sont remplacés par leur borne supérieure (`Object` par défaut)

    `void add(E x)` => `void add(Object x)`
  - Génération de cast si nécessaire

```
Rectangle r = lrects.get(i);
   => Rectangle r = (Rectangle)lrects.get(i);
```

- Objectif
  - Compatibilité avec les versions antérieures

Pour conclure

- Mais attention!
  - à l'exécution, il n'existe en fait qu'une seule « version » de la classe générique : son type brut
  - Eviter les variables de classe sur les génériques: un seul exemplaire au final « partagé » dans le type brut
  - code mixte possible, mais attention le compilateur accepte (moyennant warnings):

```
Liste lbrute = lrects;
lbrute.add(new Compte());
```