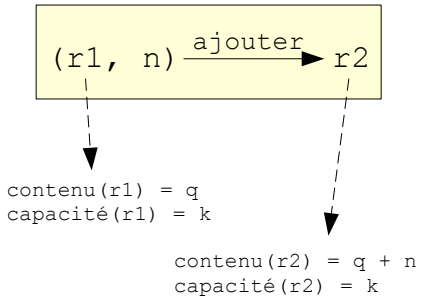
Méthodologie


```
TDA RESERVOIR
 définit R ; utilise N, B
 opérations
  ajouter  : R × N -/-> R
  capacité :      R ---> N
  contenu  :      R ---> N
  créer    :      N -/-> R
  plein    :      R ---> B
  retirer  : R × N -/-> R
  vide     :      R ---> B
 préconditions
  pre-ajouter(r, n) : contenu(r) + n ≤ capacité(r)
  pre-créer(n)      : n > 0
  pre-retirer(r, n) : contenu (r) ≥ n
 axiomes
  contenu(r) ≤ capacité(r)
  plein(r) = (contenu(r) = capacité(r))
  vide(r) = (contenu(r) = 0)
  contenu(créer(n)) = 0
  capacité(créer(n)) = n
  contenu(ajouter(r, n)) = contenu(r) + n
  capacité(ajouter(r, n)) = capacité(r)
  contenu(retirer(r, n)) = contenu(r) - n
  capacité(retirer(r, n)) = capacité(r)
```

TDA (type de données abstrait) : ensemble d'éléments muni d'opérations définies par une spécification qui exprime la sémantique de ces opérations.

voir cours POO1 de L2



pour accéder à **l'état** d'un réservoir

pour accéder à un **réservoir**

requête / *générateur*

fonction *partielle* / fonction *totale*

$E_{définition} \subset E_{départ}$

$E_{définition} = E_{départ}$

requête *de base* / requête *dérivée*

requêtes "**indispensables**"

requêtes **calculables** à partir des requêtes de base

créateur / *commande*

pour accéder à un réservoir à partir d'une **valeur externe**

pour accéder à un réservoir à partir d'un **autre réservoir**

Méthodologie

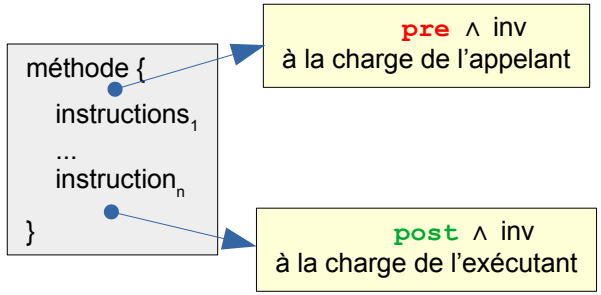
Programmation par contrat (B. Meyer) :
Méthode de construction logicielle
qui conçoit les composants d'un système
de telle façon qu'ils coopèrent
sur la base de contrats définis de façon formelle.

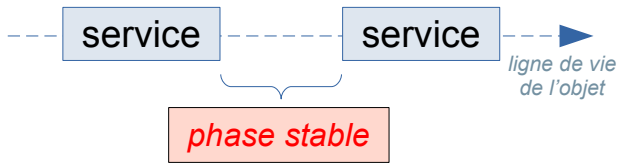
```
/**
 * @inv
 *      capacity() > 0
 *      content() >= 0
 *      content() <= capacity()
 *      empty() == (content() == 0)
 *      full() == (content() == capacity())
 * @cons
 *      $ARGS$ int n
 *      $PRE$  n > 0
 *      $POST$ capacity() == n
 *             content() == 0
 */
interface Tank {
    // REQUETES
    int capacity();
    int content();
    boolean full();
    boolean empty();
    // COMMANDES
    /**
     * @pre
     *      q >= 0
     *      content() + q <= capacity()
     * @post
     *      content() == old content() + q
     *      capacity() == old capacity()
     */
    void add(int q);
    /**
     * @pre
     *      q >= 0
     *      content() >= q
     * @post
     *      content() == old content() - q
     *      capacity() == old capacity()
     */
    void remove(int q);
}
```

spécification du TDA `RESERVOIR` intégrée dans l'interface `Tank` sous forme d'invariant et de contrats

Invariant de type : spécification qui décrit "ce qui est toujours vrai" par rapport à l'état observable de toutes les instances d'un même type lorsqu'elles sont en phase stable.

Contrat : spécification de constructeur ou de méthode qui en définit les conditions de bonne utilisation ainsi que l'effet produit par son exécution.





Méthodologie

Une *classe correcte* doit être :
- spécifiée par un TDA
- codée comme un module

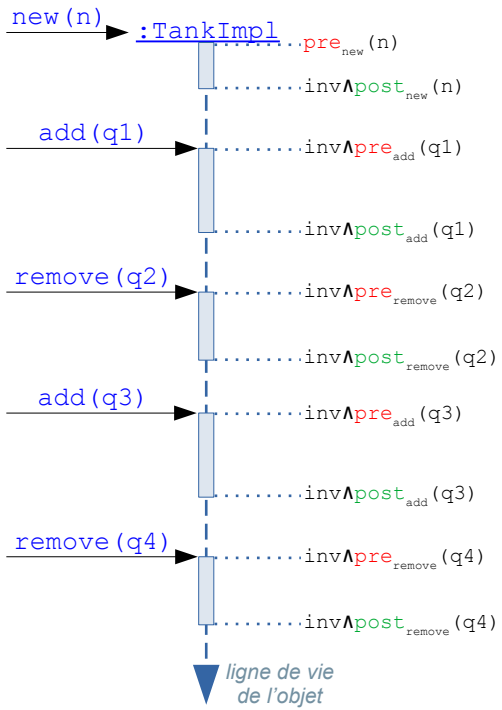
Dans une classe correcte :

- tout appel à un constructeur `k` respecte
$\{pre_k(x)\}$ `new k(x)` $\{inv \wedge post_k(x)\}$
- tout appel à un service `m` respecte
$\{inv \wedge pre_m(x)\}$ `o.m(x)` $\{inv \wedge post_m(x)\}$
- tout appel à une méthode auxiliaire `p` respecte
$\{pre_p(x)\}$ `o.p(x)` $\{post_p(x)\}$

contient la description des contrats et de l'invariant

```
public class TankImpl implements Tank {
    // ATTRIBUTS
    ... attributs privés
    // CONSTRUCTEURS
    public TankImpl(int n) {
        // doit réaliser son contrat
        // et établir inv
    }
    // REQUETES
    public int capacity() { ... }
    public int content() { ... }
    public boolean full() { ... }
    public boolean empty() { ... }
    // COMMANDES
    public void add(int q) {
        // doit réaliser son contrat
        // tout en rétablissant inv en sortie
        ...
    }
    public void remove(int q) {
        // doit réaliser son contrat
        // tout en rétablissant inv en sortie
        ...
    }
    // OUTILS
    ... méthodes auxiliaires non publiques
}
```

Rappel (*schéma de Hoare*) : $\{P\}$ `I` $\{Q\}$
si `I` commence quand P est vraie
alors Q est vraie quand (et si) `I` termine



Méthodologie

Les requêtes doivent être des *fonctions "pures"*

pas d'effet de bord !

```
/**
 * @pre
 *      content() + q <= capacity()
 *      ...
 */
void add(int q);
```

Les préconditions des méthodes doivent être testées :
- tout au début du corps de la méthode
- par une levée gardée d'`AssertionError`

Les postconditions des méthodes (et les invariants) doivent être codés **ailleurs**.

exécutée en permanence

exécutée seulement avec l'option -ea

```
public void service() {
    Contract.checkCondition(pre);
    ... cœur de la méthode
}

private void auxMethod() {
    assert pre;
    ... cœur de la méthode
}
```

Les préconditions des constructeurs doivent être testées :
- lors du passage des paramètres à l'appel interne à un autre constructeur
- dans des méthodes statiques privées

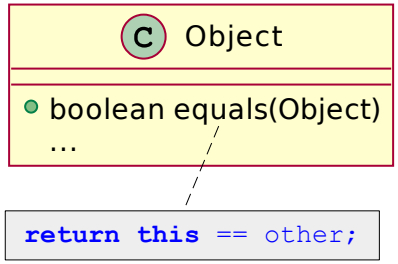
Les postconditions des constructeurs (et les invariants) doivent être codés **ailleurs**.

teste la précondition **ET** respecte l'appel initial à **`super(...)`**

```
public TankImpl(int c) {
    super(validateThenGet(c))
    ... cœur du constructeur
}

private static int validateThenGet(int c) {
    Contract.checkCondition(c > 0);
    return c;
}
```

Méthode equals : `public boolean equals(Object other)`
Représente une relation d'équivalence sur les objets.
(À n'utiliser qu'en relation avec les collections.)



Contrat :
`@pre` : rien !
`@post` : équivalence + consistante + compatible avec **`null`**

equivalence = R, S, T
R : x ~ x
S : x ~ y ⇒ y ~ x
T : x ~ y ∧ y ~ z ⇒ x ~ z

`x.equals(null) == false`

si deux objets sont `equals` ou non, ils doivent le rester :
- pour toujours (non mutables)
- tant que les données utilisées par `equals` ne varient pas (mutables)

Méthodologie

si `e` est déjà dans **`this`** : rien n'a changé & retourne **`false`**
si `e` n'est pas déjà dans **`this`** : `e` est ajouté & retourne **`true`**

```
interface Set {
    /**
     * @post
     *      (exists e2 in old this : (e == null ? e2 == null : e.equals(e2)))
     *          ==> this n'a pas changé et result == false
     *      (forall e2 in old this : (e == null ? e2 != null : !e.equals(e2))
     *          ==> e a été ajouté à this et result == true
     *      ...
     */
    boolean add(Object e);
    ...
}
```

```
boolean res = set.add(new Integer(5));
S.o.p(res + set.size());
res = set.add(new Integer(5));
S.o.p(res + set.size());
```

→ `true 1`

→ `false 1`

```
public final class Integer ... {
    @Override
    public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof Integer) {
            Integer that = (Integer) other;
            result = (this.value == that.intValue());
        }
        return result;
    }
}
```

Implémentation : une première proposition (incorrecte)

```java
class A {
    private int n;
    A(int n) { ... }
    @Override
    public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof A) {
            A that = (A) other;
            result = (this.n == that.n);
        }
        return result;
    }
}
```

ce code n'est pas bon !

```java
class B extends A {
    private char c;
    B(int n, char c) { ... }
    @Override
    public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof B) {
            B that = (B) other;
            result = super.equals(that)
                     && (this.c == that.c);
        }
        return result;
    }
}
```

ce code n'est pas bon !

```java
A u = new A(1);
B v = new B(1,'a');
S.o.p(u.equals(v));
S.o.p(v.equals(u));
```

→ **true**

→ **false**

~~symétrie~~

à travailler en autonomie

Implémentation : une deuxième proposition (incorrecte)

```
class A {
    private int n;
    A(int n) { ... }
    @Override
    public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof A) {
            A that = (A) other;
            result = (this.n == that.n);
        }
        return result;
    }
}
```

ce code n'est toujours pas bon !

```
class B extends A {
    private char c;
    B(int n, int c) { ... }
    @Override
    public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof B) {
            B that = (B) other;
            result = super.equals(that)
                     && (this.c == that.c);
        } else if (other instanceof A) {
            result = super.equals(other);
        }
        return result;
    }
}
```

ce code n'est toujours pas bon !

```
A u = new A(1);
B v = new B(1,'a');
S.o.p(u.equals(v));
S.o.p(v.equals(u));
```

→ **true**

symétrie

```
A u = new B(1,'a');
A v = new A(1);
B w = new B(1,'b');
S.o.p(u.equals(v));
S.o.p(v.equals(w));
S.o.p(u.equals(w));
```

→ **true**

→ **false**

~~transitivité~~

à travailler en autonomie

Méthodologie


```java
class A {
    private int n;
    A(int n) { ... }
    @Override
    public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof A) {
            A that = (A) other;
            result = that.canEquals(this)
                    && (this.n == that.n);
        }
        return result;
    }
    public boolean canEquals(Object other) {
        return (other instanceof A);
    }
}
```

```java
class B extends A {
    private char c;
    B(int n, char c) { ... }
    @Override
    public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof B) {
            B that = (B) other;
            result = super.equals(that)
                    && (this.c == that.c);
        }
        return result;
    }
    @Override
    public boolean canEquals(Object other) {
        return (other instanceof B);
    }
}
```

```java
class C extends A {
    // si pas de nouveaux attributs...
    C(int n) { super(n); }
    // ... alors pas de redéfinition
    // de equals ni de canEquals
}
```

Implémentation : une proposition acceptable

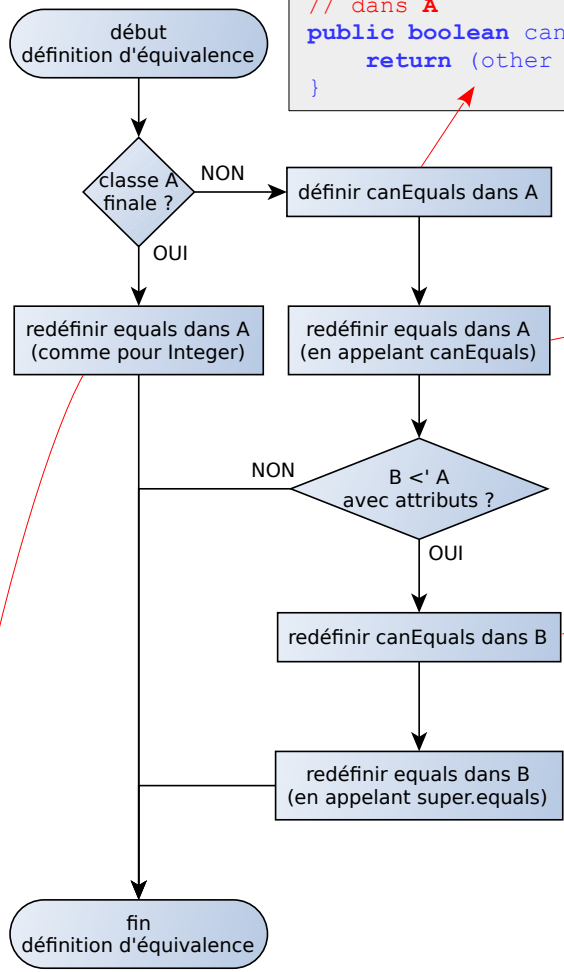
`u.equals(v)`

$$u \in A \wedge v \in A \wedge state(u) = state(v)$$
ou bien
$$u \in B \wedge v \in B \wedge state(u) = state(v)$$

symétrie + transitivité

à travailler en autonomie

Méthodologie




```java
// dans A
public boolean canEquals(Object other) {
    return (other instanceof A);
}
```

```java
// dans A
@Override
public boolean equals(Object other) {
    boolean result = false;
    if (other instanceof A) {
        A that = (A) other;
        result = that.canEquals(this)
                 && ...;
    }
    return result;
}
```

```java
// dans B extends A
@Override
public boolean canEquals(Object other) {
    return (other instanceof B);
}
```

```java
// dans B extends A
@Override
public boolean equals(Object other) {
    boolean result = false;
    if (other instanceof B) {
        B that = (B) other;
        result = super.equals(this)
                 && ...;
    }
    return result;
}
```

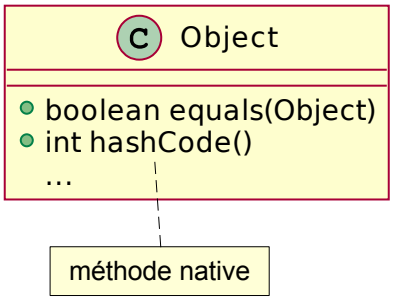
```java
// dans A
@Override
public boolean equals(Object other) {
    boolean result = false;
    if (other instanceof A) {
        A that = (A) other;
        result = ...;
    }
    return result;
}
```

Méthode hashCode : `public int hashCode()`
Représente une fonction de dispersion sur les objets, utilisée dans les tables de hachage internes à certains types de collections (`HashSet`, `HashMap`, ...).

Contrat :
`@pre` : rien !
`@post` : autant que possible, cette méthode appliquée à deux objets distincts doit retourner deux entiers distincts

Règle 1 (toute redéfinition de `equals` **nécessite** une redéfinition de `hashCode`)
`x.equals(y)` ⇒ `x.hashCode() == y.hashCode()`

Règle 2 (consistance avec `equals`)
Tant que les données utilisées par `equals` ne varient pas la valeur retournée par `hashCode` ne change pas.



Méthodologie


```java
@Override
public int hashCode() {
    final int prime = 31;
    int result = 17;
    result = prime * result + hashCodeChamp1;
    result = prime * result + hashCodeChamp2;
    ...
    return result;
}
```

champs pris en compte pour le calcul de `equals`

`hashCodeChampX`

`boolean x`	→ `(x ? 1 : 0)`
entier `x`	→ `x`
`long x`	→ `(int) (x ^ (x >>> 32))`
`float x`	→ `Float.floatToIntBits(x)`
`double x`	→ `Double.doubleToLongBits(x)` → traiter le `long` obtenu
`Object x`	→ `(x == null ? 0 : x.hashCode())`
tableau `x`	→ traiter chaque élément significatif en séquence → `Arrays.hashCode(x)` si tous les éléments sont significatifs

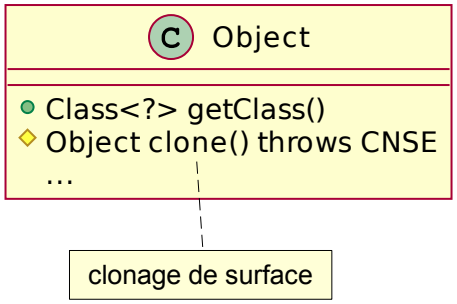
d'après Effective Java 2e éd., Joshua Bloch

à travailler en autonomie

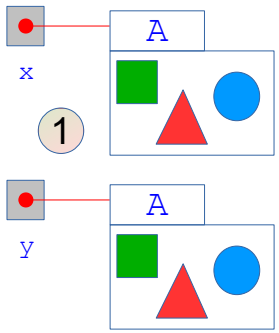
Méthodologie

Méthode clone : `protected Object clone() throws CloneNotSupportedException`

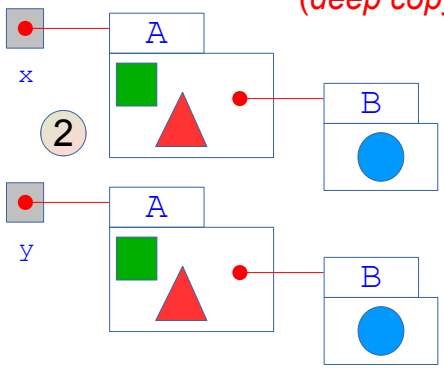
Crée et retourne une copie d'un objet.



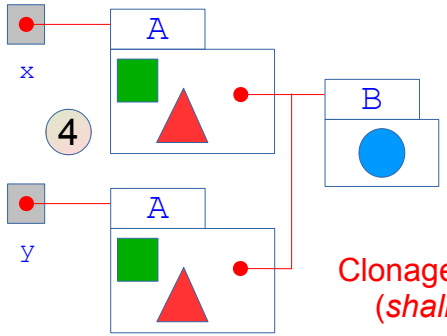
Contrat :
`@pre` : rien !
`@post` :
l'objet et son clone ne sont pas le même objet
la classe de l'objet et celle de son clone sont identiques
l'objet et son clone sont `equals`



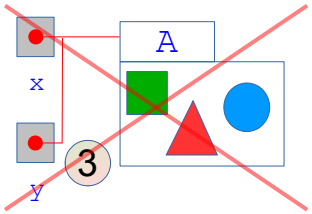
Clonage profond
(*deep copy*)



cherchez l'intrus ?



Clonage de surface
(*shallow copy*)



Héritage et clonage : une cohabitation délicate !

```
class A {
    @Override
    protected Object clone() throws CNSE {
        A clone = new A();
        // copier l'état en tant que A
        return clone;
    }
}

class B extends A {
    @Override
    protected Object clone() throws CNSE {
        B clone = new B();
        // copier l'état en tant que B
        return clone;
    }
}
```

```
class A {
    @Override
    protected Object clone() throws CNSE {
        A clone = (A) super.clone();
        ...
        return clone;
    }
}

class B extends A {
    @Override
    protected Object clone() throws CNSE {
        B clone = (B) super.clone();
        ...
        return clone;
    }
}
```

```
class A implements Cloneable {
    @Override
    protected A clone() {
        A clone = null;
        try {
            clone = (A) super.clone();
        } catch (CNSE e) {
            throw new AssertionError();
        }
        // copier l'état A de this dans clone
        return clone;
    }
}

class B extends A {
    @Override
    protected B clone() {
        B clone = (B) super.clone();
        // copier l'état B de this dans clone
        return clone;
    }
}
```

CNSE

si tout le monde a le droit de cloner
protected → public

1. implémenter `Cloneable` et capturer l'exception
2. appeler `Object.clone()` et restreindre le type de retour
3. éventuellement :
   - passer à public
   - cloner en profondeur

uniquement si clonage en profondeur