CPS — Programmation par composants

© 2019- Jacques Malenfant

Master informatique, spécialité STL – UFR 919 Ingénierie
Sorbonne Université

Jacques.Malenfant@lip6.fr

Cours 2

Programmation par composants séquentialisée

Principaux concepts à approfondir suite au premier cours

- **Composant** : objet Java matérialisant les éléments d'une application à base de composants en BCM.
  - Toutes leurs classes Java étendent directement ou indirectement la classe abstraite `AbstractComponent`.
- **Interface de composants** : interface Java marquée comme telle représentant les services offerts ou requis par des composants.
  - Toutes ces interfaces étendent indirectement `ComponentInterface`.
- **Port** : objet Java matérialisant les points d'entrée ou de sortie des appels entre les composants clients (port sortants) et les composants fournisseurs (ports entrants).
  - Toutes leurs classes étendent directement ou indirectement `AbstractPort` et implantent (au sens Java) une interface de composants.
- **Connecteur** : objet Java matérialisant la connection entre un port sortant et un port entrant.
  - Toutes leurs classes étendent directement ou indirectement `Abstract-Connector` et implantent (au sens Java) une interface de composants.

Plan

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre la différence de niveaux entre langage d'implantation de BCM et langage de programmation des applications en BCM.
   - Comprendre comment sont représentés les composants en BCM4Java.
   - Comprendre les principales parties d'un composant.
   - Comprendre comment les services sont implantés et comment les appels de services faits à un composant sont exécutées.
2. Compétences à acquérir
   - Savoir développer un composant simple en BCM4Java.
   - Savoir distinguer le code de l'application du code qui gère les composants, et savoir où il est admis d'utiliser ce dernier.
   - Savoir implanter les services et activités propres à un composant.
   - Savoir lier les appels de services via un port entrant à son implantation Java dans le composant et à une exécution sur un fil d'exécution du composant.
   - Savoir utiliser les fils d'exécution d'un composant pour faire exécuter des services et des activités propres.

Représentation d'un composant

- En BCM4Java, un composant est défini par une classe et il est créé par la méthode statique `AbstractComponent#createComponent`.
  - C'est un objet particulier qu'on ne manipule qu'à travers les opérations fournies par le *framework* BCM4Java.
- La *marque* indiquant qu'un objet représente un composant est que sa classe de définition étend `AbstractComponent` qui elle-même implante l'interface `ComponentI` :

```
// définition de composants
public class MonComposant extends AbstractComponent {
  protected MonComposant(int ndT, int nbST) { super(nbT, nbST); }
  ... // définition internes au composant
}

// creation d'un composant dans le code framework
String uri = AbstractComponent.createComponent(
                MonComposant.class.getCanonicalName(), new Object[]{1, 0});
```

  - Les paramètres du constructeur (par défaut) indiquent le nombre de fils d'exécution que le composant aura à sa disposition pour exécuter ses services (nous y reviendrons plus tard).

Principales parties d'un composant

- La définition d'un composant comporte tout ce qu'une classe Java peut contenir, mais avec caractéristiques spécifiques :
  - **Partie classes et types internes** : comme pour les objets Java, la classe de définition d'un composant peut déclarer des classes et types internes (*inner*) pour ses composants (instances).
  - **Partie variables et constantes** : elle peut aussi déclarer des constantes et des variables.
  - **Partie constructeurs** : `AbstractComponent` définit des constructeurs par défaut mais les composants peuvent définir leurs propres constructeurs (qui appellent les précédents).
  - **Partie cycle de vie** : méthodes redéfinies d'`AbstractComponent` précisant le comportement du composant au fil de sa vie (démarrage, exécution, finalisation, arrêt).
  - **Partie implantation des services** : méthodes implantant les différents services du composant.
  - **Partie méthodes auxiliaires** : méthodes internes, utilisées par les autres méthodes et qui ne correspondent pas directement à des implantation de services.

Interfaces requises versus offertes

- Un composant requiert et offre des services en requérant et offrant des interfaces de composants.
- Les interfaces de composants déclarent les signatures d'appels de services qui indiquent comment un composant client doit appeler les services du composant serveur.
  - L'interface requise donne les signatures d'appels qui doivent être utilisées dans le code du composant client pour appeler les services via son port sortant.
  - L'interface offerte donne les signatures utilisées pour appeler les services du composant serveur via son port entrant.
- C'est le rôle du connecteur de faire la liaison entre les deux, comme on l'a vu dans notre premier exemple complet : l'appel sur la signature `sum` de l'interface requise `SummingServicesCI` devient dans le connecteur `SummingConnector` un appel sur la signature `add` de l'interface offerte `CalculatorServicesCI`.

Implantation des services offerts par des méthodes

- Les interfaces de composants offertes ne donnent qu'une *vision externe* des services proposés par le composant :
  - Le composant ***n'implante pas au sens Java ses interfaces de composants offertes*** et il n'est pas forcé d'implanter des méthodes de même signature que celles qui y sont proposées.
  - Il peut définir des méthodes d'implantation de services de signatures différentes, ou même d'implanter un service grâce à plusieurs méthodes.
- C'est le rôle du port entrant de faire la liaison entre les deux, comme on l'a vu dans notre premier exemple complet : l'appel sur la signature `add` de l'interface offerte `CalculatorServicesCI` devient dans le port entrant `CalculatorServicesInboundPort` un appel à la méthode d'implantation `addService` du composant `Calculator`.

Exécution d'un service ou d'une tâche I

- Les composants BCM ont leurs propres fils d'exécution (*threads*) qui seront utilisés pour exécuter :
  - des requêtes correspondant à des appels de service faits par des clients et passant par les ports entrants,
  - des tâches correspondant à des activités que le composant peut lancer de lui-même.
- Les fils d'exécution (*threads*) des composants sont gérés en groupes (*pools*) définis par les classes d'`ExecutorService` de Java. Ces classes implantent deux méthodes principales pour leur soumettre du code à exécuter :
  - `<T> Future<T> submit(Callable<T> request)`
  - `Future<?> submit(Runnable task)`
- Qu'est qu'un `Callable` ? un `Runnable` ?
  - `Callable<T>` est une interface exigeant une implantation de la méthode `<T> call()` $\Rightarrow$ une *requête* avec résultat.

Exécution d'un service ou d'une tâche II

- `Runnable` est une interface exigeant une implantation de la méthode `void run()` $\Rightarrow$ une *tâche* sans résultat.

- Pour s'abstraire de l'implantation précise des fils d'exécution, les composants BCM offrent une méthode principale pour soumettre des requêtes à leurs fils d'exécution :
  - `<T> T handleRequest(ComponentService<T> request)` : appel *synchrone* qui soumet une requête exécutée dès qu'un fil d'exécution se libère et où le fil d'exécution du client est bloqué en attente du résultat jusqu'au retour de ce dernier.

  et une méthode principale pour leur soumettre une tâche :
  - `void runTask(ComponentTask task)` : appel *asynchrone* qui soumet une tâche exécutée dès qu'un fil d'exécution se libère et où le fil d'exécution du client poursuit immédiatement son exécution sans attendre de résultat.

Les requêtes et les tâches des composants

- BCM étend les `Callable<T>` de Java par l'interface suivante définie dans `ComponentI` :

```
public interface ComponentService<V> extends Callable<V> {
  public void setOwnerReference(ComponentI owner);
  public ComponentI getServiceOwner();
  public Object getServiceProviderReference();
}
```

`AbstractService` propose une implantation à étendre ; les détails seront examinés plus loin, mais le détenteur (*owner*) est le composant exécutant la requête et il est automatiquement affecté par ce dernier lors de la soumission de la requête.

- De même pour les tâches, BCM étend les `Runnable` de Java par l'interface suivante :

```
public interface ComponentTask extends Runnable {
  public void setOwnerReference(ComponentI owner);
  public ComponentI getTaskOwner();
  public Object getTaskProviderReference();
}
```

`AbstractTask` propose une implantation à étendre ; les détails seront également examinés plus loin (idem).

Appels passés à un composant serveur et exceptions I

- Les modes d'appels aux composants modifient le comportement vis-à-vis des exceptions selon la sémantique des méthodes `submit` et la manière de récupérer les résultats.
- Avant même toute exécution de requête ou de tâche, les méthodes `submit` peuvent lancer deux exceptions :
  - `RejectedExecutionException` : si la tâche n'a pas pu être démarrée (politique de gestion de la surcharge).
  - `NullPointerException` : si le paramètre requête est `null`.
- Les exceptions `e` lancées durant l'exécution d'une requête par `handleRequest` sont relayées par cette dernière et peuvent être rattrapées comme `ExecutionException` ayant pour cause `e`.
  - Pour faire exécuter une requête sur un service `r` de type de retour `int`, dans un port entrant d'un composant de type `C`, voici les idiomes usuels :

Appels passés à un composant serveur et exceptions II

```
return this.getOwner().handleRequest(
        new AbstractComponent.AbstractService<Integer>() {
          public Integer call() {
            return ((C)this.getServiceProvider()).r(...);
        });
// avec lambda-expressions depuis Java 8
return this.getOwner().handleRequest(o -> ((C)o).r(...));
```

- pour faire exécuter une requête `t` avec type de retour `void` :

```
return this.getOwner().handleRequest(
        new AbstractComponent.AbstractService<Void>() {
          public Void call() {
            ((C)this.getServiceProvider()).t(...);
            return null;
        });
// avec lambda-expressions depuis Java 8
return this.getOwner().handleRequest(
                o -> {((C)o).t(...); return null;});
```

Appels passés à un composant serveur et exceptions III

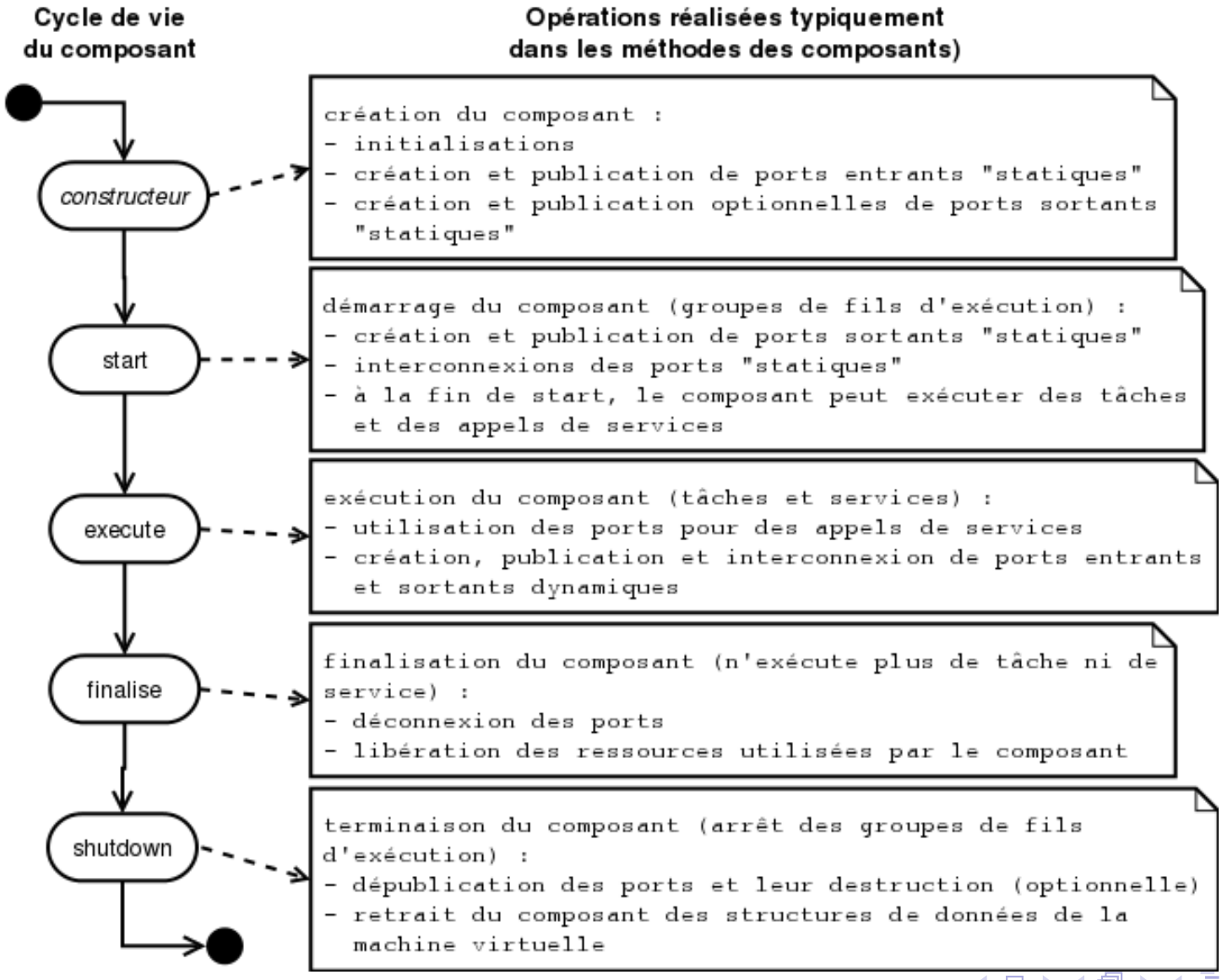
- Pour les tâches exécutées de manière asynchrone par `runTask`, impossible de propager les exceptions à l'appelant.

  Les exceptions sont alors silencieuses à moins de provoquer explicitement l'impression de la pile d'exécution.

  - Pour faire exécuter un service comme une tâche `a` de type de retour `void` tout en provoquant cette impression, voici les idiomes à adopter la plupart du temps :

```
this.getOwner().runTask(
        new AbstractComponent.AbstractTask() {
          public void run() {
            try { ((C)this.getTaskOwner()).a(...);
            } catch(Exception e) { e.printStackTrace(); }});
// avec lambda-expressions depuis Java 8
this.getOwner().runTask(
        o -> { try { ((C)o).a(...);
              } catch(Exception e) { e.printStackTrace(); }});
```

Cycle de vie des composants



Création statique de composants

- Pour éviter la confusion entre appels d'opérations de la machine virtuelle BCM et code de l'application à base de composants, BCM4Java masque autant que possible la référence aux objets Java qui représentent les composants.
- Pour arriver à cela, tous les constructeurs des classes définissant des composants *doivent être déclarés* `protected`, ce qui empêche de les appeler ailleurs que dans le composant lui-même.
- La création d'une instance de composant se fait par l'appel à la methode statique `AbstractComponent#createComponent` :

```
AbstractComponent.createComponent(
    URIProvider.class.getCanonicalName(), // nom de la classe à instancier
    new Object[]{...})                    // tableau des paramètres du constructeur
```

  Cette méthode retourne une URI propre au nouveau composant, qui peut être utilisée pour faire des opérations sur ce composant ; nous y reviendrons.

BCM et BCM4Java

- BCM pourrait être implanté dans différents langages à objets mais la seule implantation actuelle est en Java.
- BCM4Java utilise Java à la fois comme langage d'implantation de BCM et comme langage d'écriture des programmes en BCM.
- Plus précisément, BCM4Java se présente comme un *framework* implantant en Java les composants et le langage dans lequel ces composants sont programmés (ses services) est aussi Java.
- Il faut donc bien distinguer les opérations sur les entités de BCM, où on utilise le *framework*, du code des applications où on utilise simplement Java pour manipuler des données de base :

```
doPortDisconnection(myPort.getPortURI()); // ⇒ framework
ArrayList<String> l; l.add("un"); // code Java d'un composant
```

- Les opérations du *framework* ne doivent être utilisées que dans des contextes *légitimes* :
  - dans le code d'*assemblage* et de *déploiement* des composants
  - et dans chaque composant pour les *opérations sur lui-même*.

Plan

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre le rôle des interfaces de composants.
   - Comprendre comment sont représentées les interfaces de composants en BCM4Java.
   - Comprendre les principaux types d'interfaces de composants définies dans BCM et leurs interrelations.
   - Comprendre comment les interfaces de composants doivent être utilisées en relation avec les autres entités BCM (composants, ports, connecteurs, ...).
2. Compétences à acquérir
   - Savoir définir des interfaces de composants requises et offertes.
   - Savoir utiliser les deux principaux patrons de conception sur les interfaces de composants en BCM :
     - Interfaces à la fois requises et offertes.
     - Définition partagée d'interfaces d'implantation des services et interfaces de composants offertes.

Comment sont représentées les interfaces de composants ?

- Les interfaces de composants définissent les signatures d'appel des services, que ce soient celles utilisées par le client (requises) ou celles acceptées par le fournisseur (offertes).
- Elles sont représentées par des interfaces Java, mais pour les différencier des simples interfaces Java, elles étendent toutes indirectement l'interface `ComponentInterface`, ce qui les étiquette comme interfaces de composants.
  - Aucune interface de composants définie par les utilisateurs ne doit étendre *directement* `ComponentInterface` !
  - Toutes les interfaces de composants sont soit offertes, soit requises soit les deux et ceci est marqué par le fait qu'elles étendent directement ou indirectement les interfaces `OfferedCI`, `RequiredCI` voire les deux.
  - `OfferedCI` et `RequiredCI` étendent `ComponentInterface`.

L'interface `OfferedCI` et les interfaces offertes

- Toutes les interfaces de composants offertes étendent directement ou indirectement `OfferedCI`.
- Elles définissent les signatures de services appelables sur des composants jouant alors le rôle de fournisseurs.
- Puisqu'elles s'appliquent à des objets potentiellement appelables en RMI :
  - `OfferedCI` étend `java.rmi.Remote` ;
  - toutes les méthodes doivent pouvoir lancer `RemoteException`.
- Les interfaces de composants offertes sont implantées (au sens Java) par les ports entrants (*inbound*) qui matérialisent les points d'entrée pour les appels vers les composants fournisseurs de services.
- Les connecteurs appellent les ports entrants selon les signatures définies par l'interface de composants offerte correspondante.

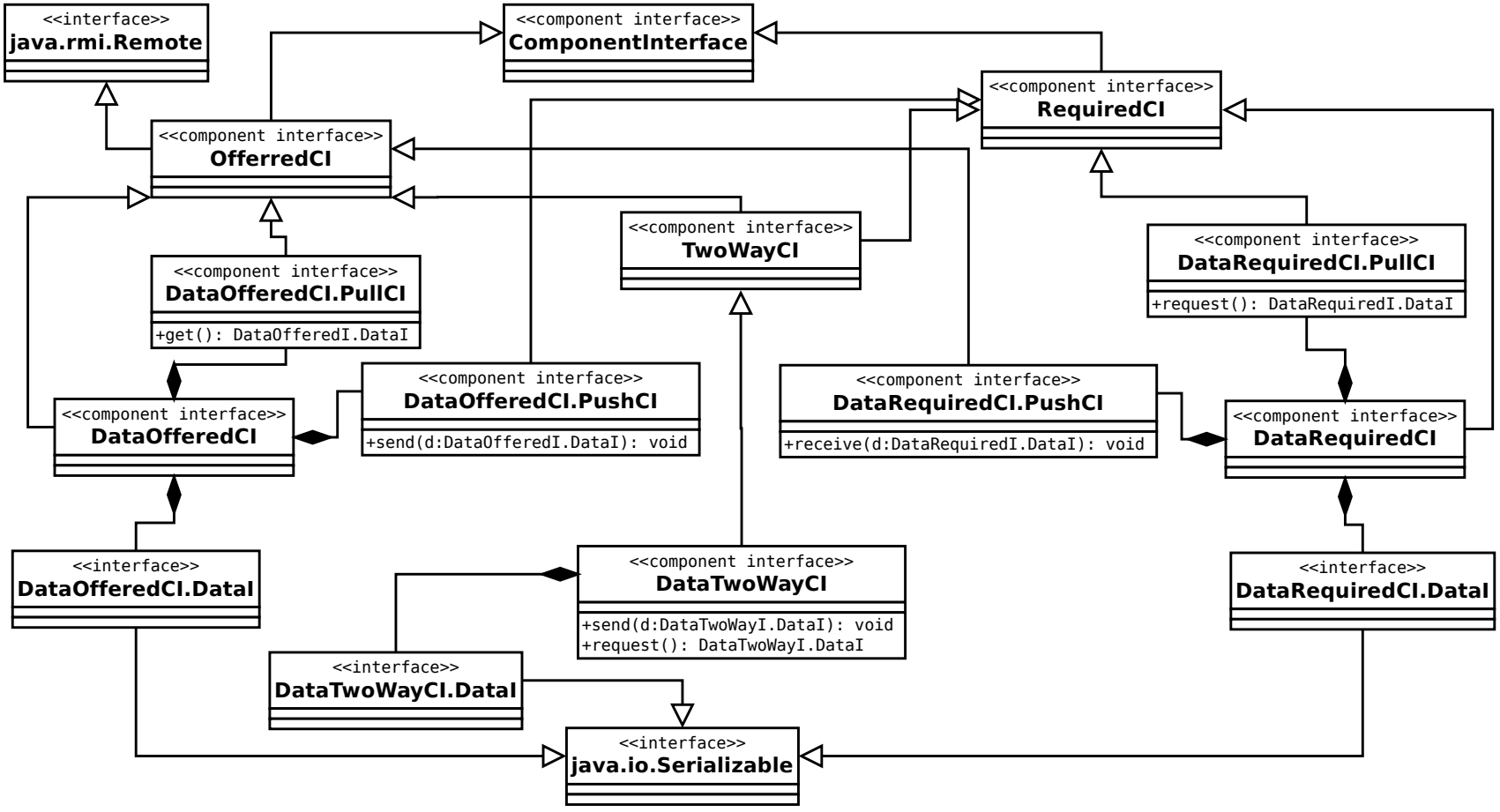
L'interface `RequiredCI` et les interfaces requises

- Toutes les interfaces de composants requises étendent directement ou indirectement `RequiredCI`.
- Elles définissent les signatures de services que les composants souhaitent appeler, ces composants jouant alors le rôle de clients.
  - Elles permettent d'écrire le code du composant appelant *sous l'hypothèse* de signatures de services attendues du fournisseur de services.
- Les interfaces de composants requises sont implantées (au sens Java) par les ports sortants (*outbound*) qui matérialisent les points de sortie pour les appels depuis ces composants clients de services.
- Elles sont aussi implantées (au sens Java) par les connecteurs qui sont appelés selon les signatures des services requis par le port sortant du client.

Conformité entre interfaces requises et offertes

- Informellement, une interface offerte est ***conforme*** à une interface requise R si elle déclare toutes les méthodes de R.
  - Plus formellement, pour toute méthode apparaissant dans l'interface requise, il existe une méthode de même nom, avec le même nombre de paramètres qui ont entre eux des types conformes et des types de résultats conformes au sens de Java.
  - L'interface offerte peut donc contenir plus de signatures que l'interface requise tout en restant conforme.
- Des interfaces requises et offertes conformes peuvent toujours mener à une connexion correcte.
  - Mais quand une interface offerte n'est pas conforme à une interface requise, il est parfois possible de les rendre conformes modulo quelques changements (noms de méthodes, ordre des paramètres, retrait ou ajout de paramètres réels par défaut, *etc.*).
  - Ce genre d'adaptations d'interfaces peuvent être programmées dans un connecteur.

Autres types d'interfaces prédéfinies de BCM



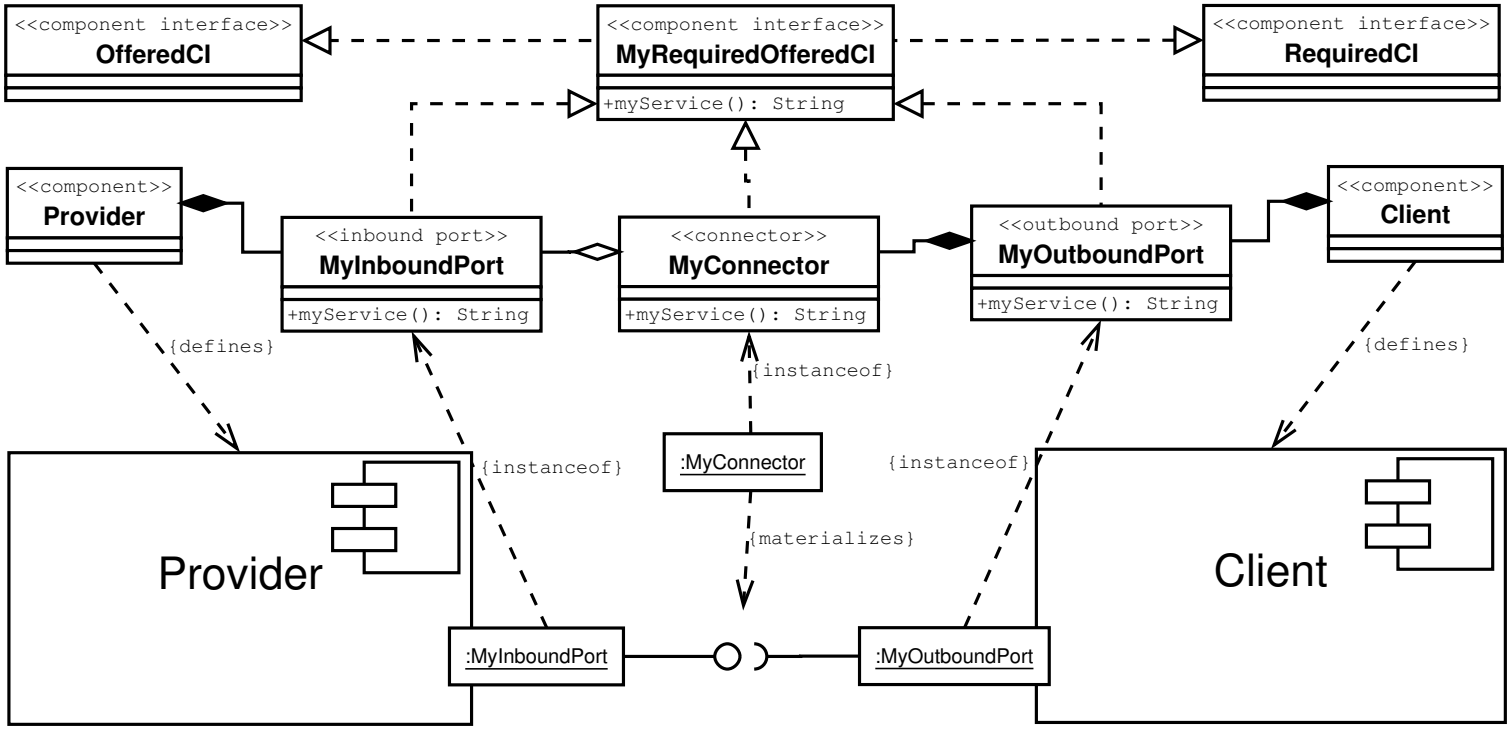
- Note : dans les sources de BCM, le répertoire `examples` contient un exemple appelé `pingpong` qui illustre l'utilisation de tous les types d'interfaces de BCM.

Exemples d'interfaces de composants

```
package fr.sorbonne_u.alasca.summing.vectorsummer.interfaces;
import fr.sorbonne_u.components.interfaces.RequiredCI;

public interface SummingServicesCI
extends RequiredCI
{
  public double sum(double x, double y) throws Exception;
}
```

```
package fr.sorbonne_u.alasca.summing.calculator.interfaces;
import fr.sorbonne_u.components.interfaces.OfferedCI;

public interface CalculatorServicesCI
extends OfferedCI
{
  public double add(double x, double y) throws Exception;
  public double subtract(double x, double y) throws Exception;
}
```

Patron d'interface à la fois offerte et requise



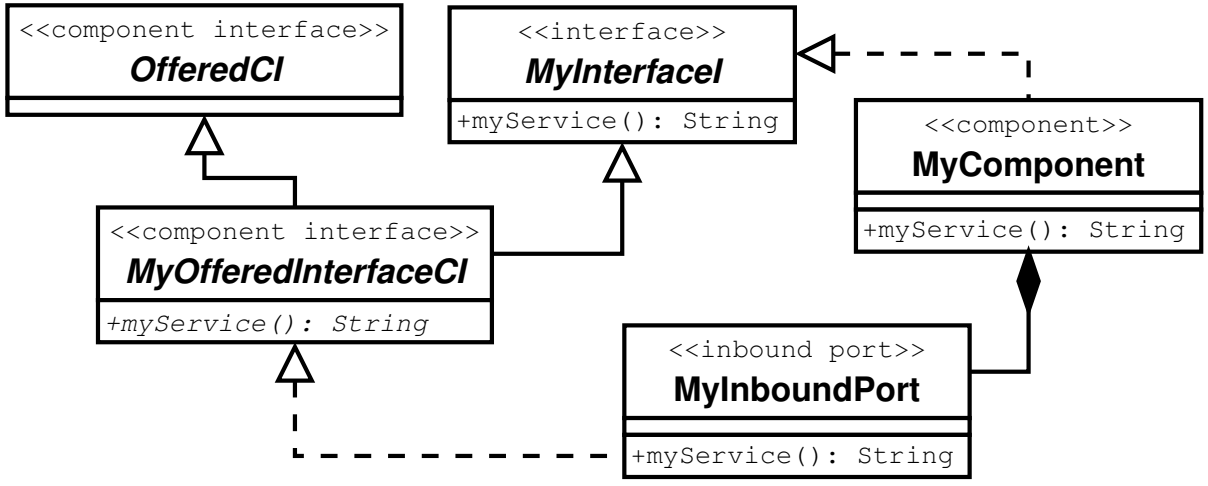
- Ici, l'interface `MyRequiredOfferedCI` étend à la fois `OfferedCI` et `RequiredCI`, ainsi elle est à la fois offerte et requise, et donc elle est implantée par les deux ports et le connecteur à l'identique.
- Utilisable lorsque le développeur maîtrise à la fois les composants fournisseurs et clients, pour leur imposer la même interface.

Exemple d'interface à la fois offerte et requise

```
package fr.sorbonne_u.alasca.summing.calculator.interfaces;
import fr.sorbonne_u.components.interfaces.OfferedCI;
import fr.sorbonne_u.components.interfaces.RequiredCI;

public interface CalculatorServicesCI
extends RequiredCI, OfferedCI {
 public double add(double x, double y) throws Exception;
 public double subtract(double x, double y) throws Exception;
}
```

Patron de définition partagée d'interface offerte et implantée

- Problème : un composant doit définir des services correspondant à l'interface offerte mais *ne doit jamais* l'implanter (au sens Java).
- Si on souhaite avoir les mêmes signatures de méthodes dans le composant que dans l'interface de composants offerte, il faut les factoriser dans une interface Java commune.



- **Rappel important** : la classe décrivant le composant *ne doit jamais* implanter l'interface `MyOfferedInterface`, sinon son instance sera considérée comme un port par le *framework* BCM.

Exemple d'interface partagée implantée et offerte

```
package fr.sorbonne_u.alasca.summing.calculator.interfaces;
public interface CalculatorServicesImplementationI
{
 public double add(double x, double y) throws Exception;
 public double subtract(double x, double y) throws Exception;
}
```

```
package fr.sorbonne_u.alasca.summing.calculator.interfaces;
import fr.sorbonne_u.components.interfaces.OfferedCI;
import fr.sorbonne_u.components.interfaces.RequiredCI;
public interface CalculatorServicesCI
extends CalculatorServicesImplementationI, RequiredCI, OfferedCI
{
 // Répétition nécessaire pour respecter les règles RMI
 @Override
 public double add(double x, double y) throws Exception;
 @Override
 public double subtract(double x, double y) throws Exception;
}
```

Plan

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre le rôle des ports, des connecteurs et des registres.
   - Comprendre comment sont représentés les ports et les connecteurs.
   - Comprendre les principaux types de ports et de connecteurs définis dans BCM et leurs interrelations.
   - Comprendre comment les ports doivent être utilisées en relation avec les autres réalisations en BCM des concepts de composants et les registres.
2. Compétences à acquérir
   - Savoir définir des ports entrants et sortants.
   - Savoir utiliser les ports sortants dans des appels d'un composant client vers un composant fournisseur.
   - Savoir programmer des connecteurs simples et des connecteurs rendant des interfaces requises et offertes compatibles.
   - Savoir programmer des ports entrants pour relayer les appels vers un service implanté dans le composant, en gérant correctement les fils d'exécution de ce dernier.

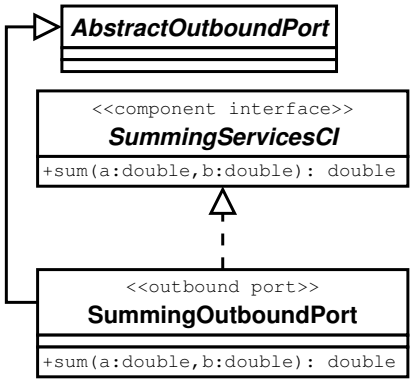
Les ports en BCM

- Les ports sont des objets Java, créés à partir de classes spécifiques, et détenus par les composants.
- Avant l'utilisation, un port :
  - doit avoir son interface de composants ajoutée à la liste des interfaces requises ou offertes du composant ;
  - est ensuite créé comme un objet Java, en lui passant (entre autres) la référence au composant qui le détient (son *owner*) ;
  - est publié dans les registres pour la connexion ;
  - et enfin connecté à un autre port.
- Après l'utilisation, le port doit être déconnecté puis dépublié (obligatoire avant la récupération du composant) puis détruit (optionnel, sinon cela se fait automatiquement lors de la destruction du composant).
- *Attention, le port et sa référence d'objet Java ne doivent jamais être utilisés à l'extérieur du composant qui le détient.*

Les ports sortants

- Les ports sortants sont détenus par les composants clients.
- Côté composant client, leur principal rôle est de permettre de faire un appel de service selon l'interface requise, appel qui sera relayé au composant fournisseur grâce à la connexion.
- Ils sont créés par une expression `new`, comme n'importe quel objet Java.
- Ils sont mémorisés par le composant et on peut le retrouver leur référence grâce à leur URI par le méthode `findPortFromURI` d'`AbstractComponent`.
- Pour utiliser un port dans le composant, il est toutefois souvent plus simple de conserver la référence sur un port retournée par l'expression `new` dans une variable du composant.
- Leur connexion se fait par le composant qui les détient à l'aide de l'URI du port entrant avec lequel ils doivent se connecter grâce à la méthode `doPortConnection` (voir plus loin).

Exemple de classe définissant un port sortant




```
public class SummingOutboundPort
extends AbstractOutboundPort
implements SummingServicesCI
{
  public SummingOutboundPort(ComponentI owner) {
    // le constructeur d'AbstractOutboundPort attend
    // l'interface implantée par le port et la
    // référence au composant qui le détient.
    super(SummingServicesCI.class, owner);
  }

  @Override
  public double sum(double a, double b) {
    ((SummingServicesCI)this.getConnector()).sum(a, b);
  }
}
```

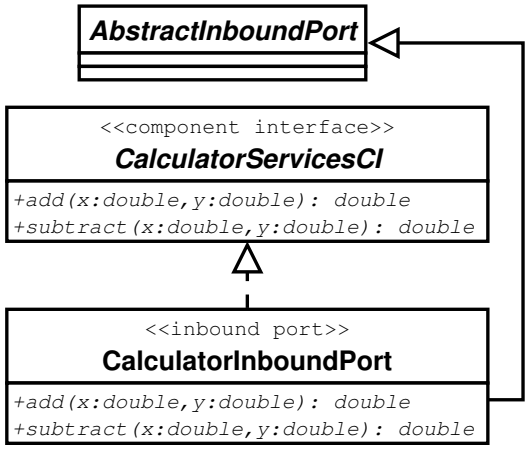
Exemple d'utilisation de port sortant

```
@RequiredInterfaces(required={SummingServicesCI.class})
public class VectorSummer extends AbstractComponent {
  protected SummingOutboundPort sop; // facilite l'utilisation dans le composant

    // ...
    this.sop = new SummingOutboundPort(this);           // création du port
    this.sop.publishPort();          // publication du port dans les registres
    // code réalisant la connexion (voir plus loin)
    // ...

  public double summingMyVector() {
    double result = 0.0;
    for(int i = 0 ; i < this.myVector.length ; i++) {
      result = this.sop.sum(result, this.myVector[i]); // utilisation
    }
    return result;
  }

    // généralement dans les méthodes finalise et shutdown:
    // déconnexion du port (voir plus loin)
    this.sop.unpublishPort();    // dépublication
    this.sop.destroyPort();      // destruction et retrait du composant
}
```

Les ports entrants

- Les ports entrants sont détenus par les composants fournisseurs.
- Ils sont créés par une expression `new`.
- Pour la connexion, ils sont généralement passifs et connectés via le port sortant correspondant.
- Côté composant fournisseur, leur principal rôle est de transmettre les appels de service selon l'interface offerte aux méthodes d'implantation du composant fournisseur pour être exécutés.
- Les composants ayant leurs propres fils d'exécution (*threads*), la transmission d'un appel au composant impose une rupture :
  - l'appel depuis le composant client jusqu'au port entrant (inclus) est exécuté par *un fil d'exécution du client* ;
  - lors du passage de l'appel au composant fournisseur, l'exécution du service est pris en charge par *les fils d'exécution du fournisseur*.
- Un port entrant doit donc être *réentrant* (*i.e.*, code supportant sans risque d'être exécuté par plusieurs fils d'exécution à la fois).

Exemple de classe de port entrant

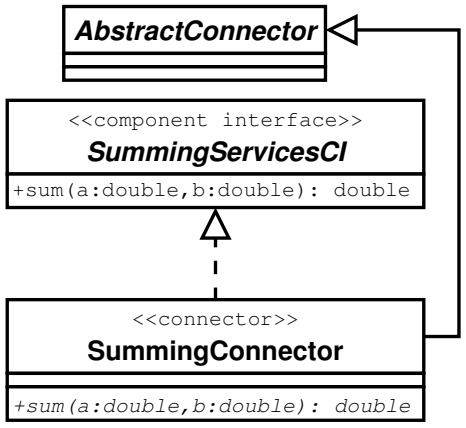



```
public class       CalculatorInboundPort
extends AbstractInboundPort implements CalculatorServicesCI
{
  public CalculatorInboundPort(ComponentI owner) {
    super(CalculatorServicesCI.class, owner);
    assert owner instanceof Calculator;
  }
  @Override      // version avec création de classe anonyme
  public double add(double a, double b) {
    return this.getOwner().handleRequest(
      new AbstractComponent.AbstractService<Double>() {
        @Override
        public Double call() {
          return ((Calculator)this.getServiceOwner()).
                                              addService(x, y);
        }
      });
  }
  @Override                  // version avec lambda de Java 8
  public double subtract(double a, double b) {
    return this.getOwner().handleRequest(
              o -> ((Calculator)o).subtractService(x, y));
  }
}
```

Exemple d'utilisation de port entrant

```
@OfferedInterfaces(offered={CalculatorServicesCI.class})
public class Calculator extends AbstractComponent {
  CalculatorInboundPort cip;      // facilite son utilisation dans le composant

    // ...
    this.cip = new CalculatorInboundPort(this);           // création du port
    this.cip.publishPort();         // publication du port dans les registres
    // ...

  public double addService(double x, double y) {
    return x + y;
  }
  public double subtractService(double x, double y) {
    return x - y;
  }

    // dans les methodes shutdown et shutdownNow...
    this.cip.unpublishPort();   // dépublication
    this.cip.destroyPort();     // destruction optionnelle
    // ...
}
```

Exemple de connecteur et de connexion/déconnexion



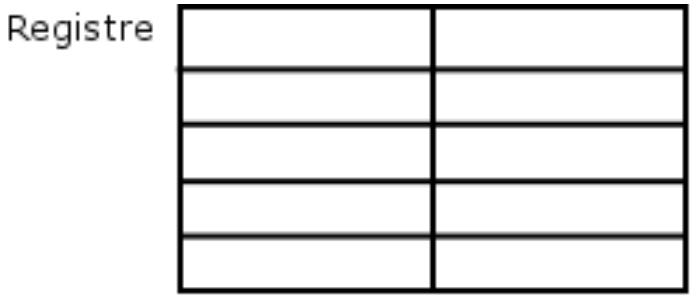

```
public class SummingConnector
extends AbstractConnector
implements SummingServicesCI
{
  @Override
  public double sum(double a, double b) {
    return ((CalculatorServicesCI)this.offering).add(a, b);
  }
}
```

- Pour connecter un port entrant à un port sortant, il faut connaître leurs URIs respectives et fournir la classe de connecteur :

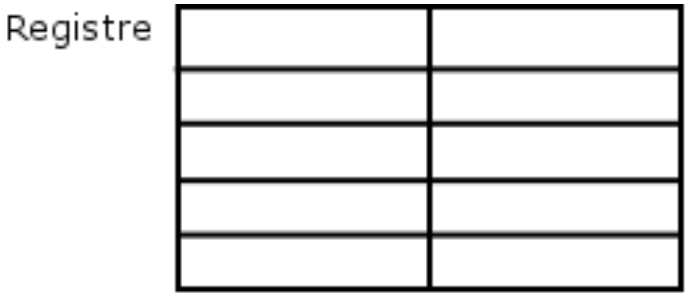
```
// Dans la classe définissant le composant VectorSummer
this.doPortConnexion(
   this.sop.getPortURI(), // URI du port sortant
   /* ici URI du port entrant, voir plus loin */,
   SummingConnector.class.getCanonicalName()); // nom de la classe
// ...
this.doPortDisconnection(this.sop.getPortURI();
```

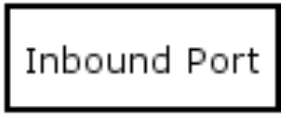
- Notez la forme utilisée pour obtenir la nom de la classe du connecteur : elle est robuste !

Déroulement des opérations de connexion (mono-JVM)



Déroulement des opérations de connexion (mono-JVM)





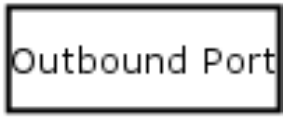

```
// Dans le composant serveur
ibp = new CalculatorServicesInboundPort("XXX", this);
```

Déroulement des opérations de connexion (mono-JVM)

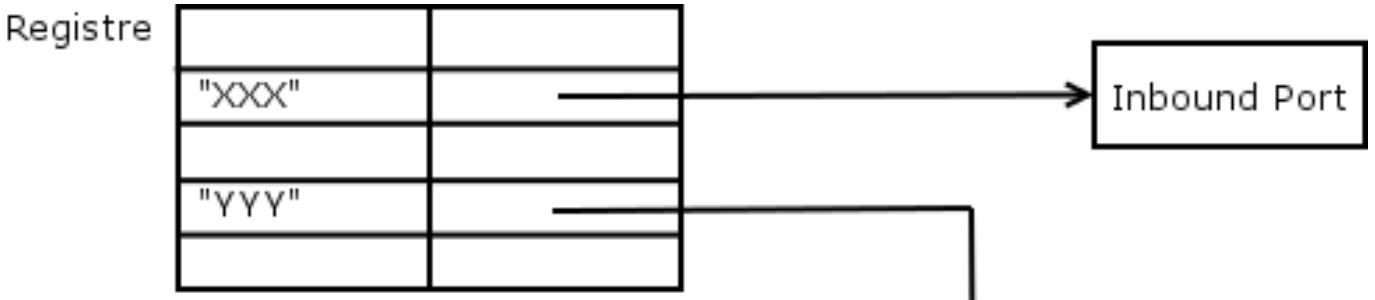

```
// Dans le composant serveur
ibp = new CalculatorServicesInboundPort("XXX", this);
ibp.publishPort();
```

Déroulement des opérations de connexion (mono-JVM)


```
// Dans le composant serveur
ibp = new CalculatorServicesInboundPort("XXX", this);
ibp.publishPort();

// Dans le composant client
obp = new CalculatorServicesOutboundPort(this);
assert obp.getPortURI().equals("YYY");  // exemple
```

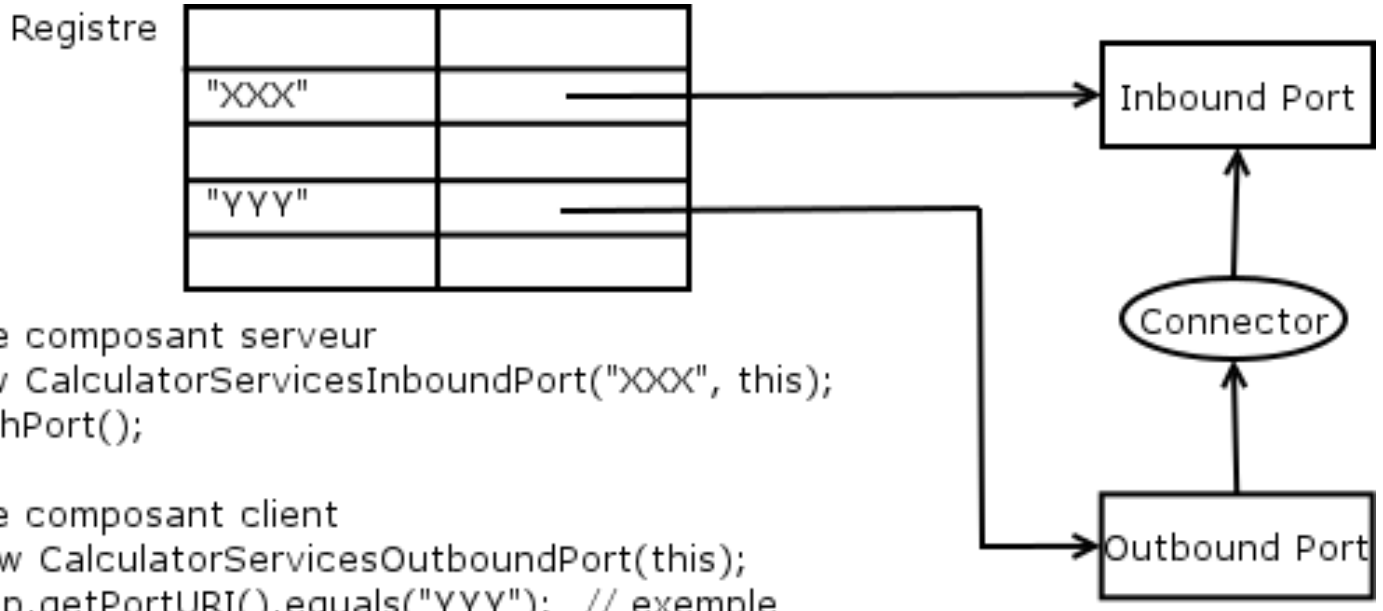


Déroulement des opérations de connexion (mono-JVM)




```
// Dans le composant serveur
ibp = new CalculatorServicesInboundPort("XXX", this);
ibp.publishPort();

// Dans le composant client
obp = new CalculatorServicesOutboundPort(this);
assert obp.getPortURI().equals("YYY");  // exemple
obp.publishPort();
```

Déroulement des opérations de connexion (mono-JVM)




```
// Dans le composant serveur
ibp = new CalculatorServicesInboundPort("XXX", this);
ibp.publishPort();

// Dans le composant client
obp = new CalculatorServicesOutboundPort(this);
assert obp.getPortURI().equals("YYY");  // exemple
obp.publishPort();

// Toujours dans le composant client
this.doPortConnection(
    obp.getPortURI(), "XXX",
   CalculatorServicesConnector.class.getCanonicalName();
```

Relations de connexion

- Les connexions sont les seuls moyens qui permettent à des composants d'échanger.
- Pour les ports et les connecteurs simples (d'interfaces dérivées directement de `RequiredCI` et `OfferedCI`) :
  - Un port sortant n'est connecté qu'à un seul port entrant.
  - Un port entrant peut être la cible de plusieurs connexions ; ceci correspond au fait qu'un composant fournisseur peut servir plusieurs composants clients, comme il se doit.
  - Cette asymétrie force les composants clients d'avoir autant de ports sortants que connexions sortantes.
  - Il y a également un connecteur par connexion.
- Pour les autres types de ports et de connecteurs (autres interfaces), les connexions sont de type 1:1, un port sortant est connecté à un seul port entrant et *vice versa*.
  - En effet, pour tous ces cas, il y a possibilité d'appel inversé de l'entrant vers le sortant (comme le mode *push* pour les port d'échange de données), ce qui force une relation 1:1.

Plan

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre la notion d'URI appliquée aux ports en BCM.
   - Comprendre le lien entre les URIs de ports et la nécessité d'avoir des registres.
   - Comprendre les différentes façons de récupérer les URIs des ports d'un composant.
2. Compétences à acquérir
   - Savoir créer un port avec une URI imposée ou avec une URI générée automatiquement.
   - Savoir connecter des ports avec des URIs imposées par le biais de partage de variables globales.
   - Savoir le faire avec un passage des URIs au moment de la création des composants.
   - Savoir connecter deux composants créés dynamiquement grâce aux opérations réflexives pour récupérer les URIs.

Qu'est-ce qu'une URI et leur rôle en BCM

- Définition : **U**nique **R**esource **I**dentifier.
- Généralisation de la notion de pointeur ou d'identifiant d'objet.
  - Quand on crée un objet Java par `new`, on récupère un identifiant d'objet qui sert à appeler cet objet.

```
Toto t = new Toto(); // crée et retourne l'identifiant affecté à t
t.m();               // l'identifiant dans t sert à appeler l'objet
```

- Mais cette référence n'a de sens que dans l'espace mémoire de la JVM où l'objet a été créé (cache une adresse mémoire).
- En programmation répartie, il faut avoir un moyen de faire référence à un objet situé dans l'espace mémoire d'une autre JVM.
- Dans BCM, les objets qu'on veut pouvoir identifier ainsi sont les ports ; ce seront les URI qui vont nous servir à cela.
- Les registres vont permettre de les connecter en faisant la correspondance entre URIs par définition globales et références d'objet locales.

Connexion et registres de publication des ports

- Tous les ports possèdent une URI (de type `String`) qui va servir à les identifier globalement pour les connexions.
  - Les URIs des ports peuvent être engendrées automatiquement à la création (cas utilisé pour les exemples précédents).
  - On peut aussi imposer une URI à un port lors de sa création en la passant à son constructeur.
  - Les ports possèdent une méthode `getPortURI()` permettant de récupérer leur URI.
- Pour lier deux ports, il faut récupérer leurs références Java, dont celle du port entrant :
  - en execution mono-JVM, on pourrait directement utiliser la référence Java obtenue lors de la création de l'objet port ;
  - mais en exécution multi-JVM, le protocole RMI va fournir une référence à un *proxy* local qui va appeler l'objet port via RMI.
- Pour avoir le même code dans les deux cas, toutes les connexions nécessitent une publication des ports dans un registre :
  - en mono-JVM, un registre local à la JVM suffit ;
  - en multi-JVM, nous verrons plus tard les registres impliqués.

Problématique de gestion des URIs

- Question : comment un composant peut-il connaître les URIs des ports *des autres composants* avec lesquels il veut se connecter ?
- Problème d'oeuf et de poule...
  - Les ports sont créés dans les composants, donc lorsqu'on est dans le composant client, on ne peut pas connaître *a priori* les URIs des ports entrants du fournisseur et *vice versa*.
- BCM fait en sorte que les ports puissent être créés avec une URI imposée ou de la générer automatiquement à la création.
- BCM fait également en sorte de conserver les informations sur les URIs de ses ports de manière à pouvoir répondre à des requêtes comme :
  - Quelles sont les URI des ports offrant ou requérant telle interface de composants ?
  - Quelles sont les interfaces offertes ou requises par un composant ?

Accès aux URIs

- Première idée : par partage de constantes globales.
  - Définir les URIs nécessaires comme des constantes globales et les imposer lors de la création des ports afin de les utiliser ensuite pour les connexions.
- Deuxième idée : passer des URI prédéterminées lors de la création des composants par les constructeurs.
  - Permet d'éviter la variable globale.
  - En mono-JVM, il est facile de générer une URI robuste.
- Solutions bien adaptées :
  - aux architectures statiques où le nombre de composants et les ports qu'ils vont créer sont connus à la conception de l'application ;
  - à certaines architectures régulières de taille fixée au déploiement en calculant les URIs et en les passant en paramètres lors de la création des composants.
- Nous verrons plus tard des techniques plus souples et plus dynamiques.

Exemple

- Alternative variable globale :

```
// dans une classe XYZ accessible globalement
public static final String URI_PORT_ENTRANT = "mon-URI";

// dans le composant fournisseur
MonInboundPort mip = new MonInboundPort(XYZ.URI_PORT_ENTRANT, this);
mip.publishPort();

// dans le composant client
MonOutboundPort mop = new MonOutboundPort(this);
mop.publishPort();
this.doPortConnection(mop.getPortURI(), XYZ.URI_PORT_ENTRANT,
                      MonConnecteur.class.getCanonicalName());
```

- Alternative par les constructeurs créant les ports :

```
String uri_port_entrant = AbstractPort.generatePortURI();
// création du composant fournisseur qui crée le port avec l'URI
AbstractComponent.createComponent(MonFournisseur.class.getCanonicalName(),
                                  new Object[]{uri_port_entrant, ...});
// création du composant fournisseur qui sauvegarde l'URI puis
// l'utilise pour la connexion
AbstractComponent.createComponent(MonClient.class.getCanonicalName(),
                                  new Object[]{uri_port_entrant, ...});
```

Activités à réaliser avant le prochain TME

1. Examiner l'exemple `internal_cs` dans l'archive BCM4Java et le comparer avec les versions du même exemple dans `basic_cs`.
2. Examiner l'exemple `pingpong` dans l'archive BCM4Java et notez les variantes dans les types de ports.