

Programmation orientée objet

Pr. ABNANE Ibtissam

Programmation orientée objet

Supposons que vous entreprenez les activités suivantes sur une base quotidienne. **Classez** cette liste dans un ordre raisonnable, puis **divisez-la en trois blocs** d'activités connexes et **donnez à chaque bloc un titre** qui résume les activités effectuées dans ce bloc.

1. Sortir du lit
2. Prendre le petit-déjeuner
3. Garer la voiture
4. S'habiller
5. Sortir la voiture du garage
6. Aller au travail en voiture
7. Découvrez ce que votre patron veut que vous fassiez aujourd'hui
8. Donnez votre avis au patron sur les résultats de la journée.
9. Faites ce que le patron veut que vous fassiez

Programmation orientée objet



Se lever :

- Sortir du lit S'habiller Prendre son petit-déjeuner

Aller au travail :

- Sortir la voiture du garage Aller au travail en voiture
- Garez la voiture

Faire le travail :

- Découvrir ce que votre patron veut que vous fassiez aujourd'hui Faire ce que le patron veut que vous fassiez
- Rétroaction au patron sur les résultats de la journée.

Programmation orientée objet

- En structurant notre liste d'instructions et en tenant compte de la structure globale de la journée (se lever, aller au travail, faire son travail), nous pouvons modifier et améliorer une section des instructions sans modifier les autres parties. Par exemple, nous pouvons améliorer les instructions pour aller au travail.....
 - Écoutez les informations sur le trafic local et la météo
 - Décidez si vous allez en bus ou en voiture.
 - Si vous allez en voiture, prenez la voiture et allez au travail.
 - Sinon, marchez jusqu'à la gare routière et prenez le bus.
- Sans se soucier de l'impact potentiel que cela pourrait avoir sur le fait de "se lever" ou de "faire son travail".

De la même manière, la structuration des programmes informatiques peut rendre chaque partie plus compréhensible et faciliter la maintenance des grands programmes.

Programmation orientée objet

- Imaginez un carnet d'adresses personnel avec quelques données stockées sur vos amis : **Nom, adresse, numéro de téléphone.**
- Citez trois choses que vous pourriez faire avec ce carnet d'adresses.
- Identifiez ensuite quelqu'un d'autre qui pourrait utiliser un carnet d'adresses identique à des fins autres que le stockage d'une liste d'amis.

Programmation orientée objet

- Avec un carnet d'adresses, on peut: trouver les coordonnées d'un ami, c'est-à-dire son numéro de téléphone, ajouter une adresse au carnet d'adresses et, bien sûr, supprimer une adresse.
- On peut créer un composant logiciel simple pour stocker les données dans le carnet d'adresses (c'est-à-dire une liste de noms, etc.) et les opérations que nous pouvons effectuer (c'est-à-dire ajouter une adresse, trouver un numéro de téléphone, etc.)
- En créant un composant logiciel simple pour stocker et gérer les adresses d'amis, On peut le réutiliser dans un autre système logiciel.
- **Ainsi, la programmation orientée objet nous permet de créer des composants logiciels réutilisables (dans ce cas, un carnet d'adresses).**

Pourquoi le POO?

- Le paradigme orienté objet **s'appuie** sur les idées qui sous-tendent le paradigme de la programmation structurée et les **étend**.
- Généralement, ce n'est pas la création du code qui est la source de la plupart des problèmes. La plupart des problèmes proviennent de :
- **Mauvaise analyse et conception** : le système informatique que nous créons ne fait pas ce qu'il faut.
- **Mauvaise maintenabilité** : le système est difficile à comprendre et à réviser lorsque, comme c'est inévitable, des demandes de changement surviennent.

Programmation orientée objet



- L'**abstraction** et l'**encapsulation** sont des principes fondamentaux qui sous-tendent l'approche orientée objet du développement de logiciels.
- L'abstraction nous permet de considérer des idées complexes tout en **ignorant les détails non pertinents** qui pourraient nous déconcerter.
- L'encapsulation nous permet de nous concentrer sur ce que fait une chose **sans tenir compte des complexités de son fonctionnement**.

Programmation orientée objet



Pensez à votre maison et imaginez que vous allez échanger votre maison pour une semaine avec un nouvel ami.

- 1- Notez trois choses essentielles que vous lui diriez sur votre maison et que vous voudriez savoir sur la sienne.
- 2- Énumérez maintenant trois détails non pertinents que vous ne diriez pas à votre ami.

Programmation orientée objet

- Vous leur donneriez probablement l'adresse, une liste de base des pièces et des équipements (par exemple, le nombre de chambres) et leur diriez comment entrer (c'est-à-dire quelle clé actionne la porte d'entrée et comment éteindre l'alarme (si vous en avez une)).
- Vous ne leur donnerez pas de détails non pertinents (comme la couleur des murs, des sièges, etc.), car cela les surchargerait d'informations inutiles.

L'abstraction nous permet de prendre en compte les détails importants de haut niveau de votre maison, par exemple l'adresse, sans nous perdre dans les détails.

Programmation orientée objet



- Pensez à votre maison et notez un article, tel qu'un téléviseur, que vous utilisez quotidiennement (et décrivez brièvement comment vous faites fonctionner cet article).
- Réfléchissez maintenant à la difficulté de décrire les composants internes de cet appareil et de donner tous les détails techniques de son fonctionnement.

Programmation orientée objet

- Il est beaucoup plus facile de décrire le fonctionnement d'un téléviseur que de décrire ses composants internes et d'expliquer en détail son fonctionnement exact. La plupart des gens ne connaissent même pas tous les composants des appareils qu'ils utilisent ni leur fonctionnement - mais cela ne les empêche pas d'utiliser des appareils tous les jours.
- Vous ne connaissez peut-être pas les détails techniques tels que le câblage des interrupteurs et leur fonctionnement interne, mais vous pouvez quand même allumer et éteindre les lumières de votre maison (et de tout nouveau bâtiment dans lequel vous entrez)
- **L'encapsulation nous permet de considérer ce que fait un interrupteur et comment nous l'utilisons, sans avoir à nous soucier des détails techniques de son fonctionnement réel.**

Programmation orientée objet

- Deux autres principes fondamentaux de l'orientation objet sont la généralisation/spécialisation (qui nous permet d'utiliser l'héritage) et le polymorphisme.
- La généralisation nous permet de considérer des catégories générales d'objets qui ont des propriétés communes, puis de définir des sous-classes spécialisées qui héritent des propriétés des catégories générales.

Programmation orientée objet



- Considérez les personnes qui travaillent dans un hôpital
- listez trois professions communes de personnes que vous vous attendez à voir employées dans cet hôpital.
- Ensuite, pour chacune de ces professions communes, citez deux ou trois catégories spécifiques de personnel.

Programmation orientée objet



Médecin :

- Médecin stagiaire, médecin junior, chirurgien, radiologue, etc.

Infirmier :

- Infirmière de triage, sage-femme, infirmière de salle d'opération.

Nettoyeur :

- Nettoyeur général Superviseur du nettoyage

Programmation orientée objet

- Maintenant que nous avons défini certaines catégories générales et certaines catégories plus spécialisées de personnel, nous pouvons considérer les éléments généraux qui sont vrais pour tous les médecins, toutes les infirmières, etc.
- Faites une déclaration sur les médecins que vous considéreriez comme **vraie pour tous les médecins**
- Faites une déclaration sur les chirurgiens qui **ne serait pas vraie pour tous les médecins.**

Programmation orientée objet

- On peut affirmer que tous les médecins ont une connaissance des médicaments, qu'ils peuvent diagnostiquer des problèmes médicaux et prescrire les médicaments appropriés.
- Pour les chirurgiens, on peut dire qu'ils savent utiliser des scalpels et d'autres équipements spécialisés et qu'ils peuvent effectuer des opérations.
- D'après la liste ci-dessus, tous les chirurgiens sont des médecins et ont donc une connaissance des conditions médicales et peuvent prescrire les médicaments appropriés. Cependant, tous les médecins ne sont pas des chirurgiens et ne peuvent donc pas tous pratiquer des opérations.
- Ce que nous spécifions comme étant vrai pour les médecins l'est également pour les médecins en formation, les jeunes médecins, etc. - ces catégories (ou classes) spécialisées peuvent hériter des attributs et des comportements associés à la classe plus générale de "médecin".

Programmation orientée objet

- La généralisation / spécialisation nous permet de définir les caractéristiques et les opérations générales d'un objet et nous permet de créer des versions plus spécialisées de cet objet. Les versions spécialisées de cet objet hériteront automatiquement de toutes les caractéristiques de l'objet plus généralisé.
- Le dernier principe sous-jacent à l'orientation objet est le polymorphisme, qui est la capacité d'interagir avec un objet dans sa catégorie généralisée, indépendamment de sa catégorie plus spécialisée.
- Faites une déclaration sur la manière dont un directeur d'hôpital peut interagir avec tous les médecins employés dans son hôpital, quel que soit le type de médecin.

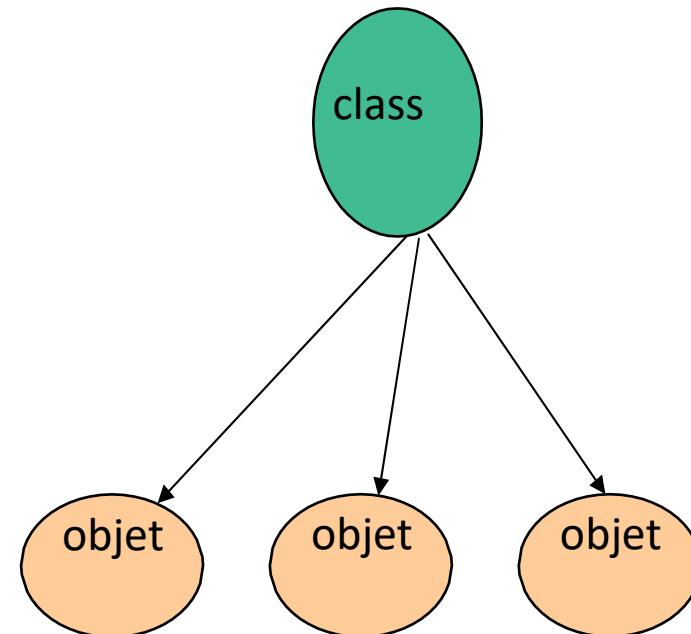
Programmation orientée objet

Un directeur d'hôpital doit payer tous les médecins (ce qui sera vraisemblablement fait automatiquement à la fin de chaque mois) et qu'il puisse prendre des mesures disciplinaires à l'encontre de tout médecin coupable de mauvaise conduite - bien entendu, cela s'applique également aux autres membres du personnel. Plus précisément, un gestionnaire pourrait vérifier que l'enregistrement médical d'un médecin est toujours en cours.

- **Il s'agit d'une tâche que la direction devrait accomplir pour tous les médecins, quelle que soit leur spécialité.**
- En outre, si l'hôpital employait un nouveau médecin spécialiste (par exemple un neurologue), sans savoir quoi que ce soit de spécifique sur cette spécialité, la direction de l'hôpital saurait quand même que a) ce personnel doit être payé et b) son inscription médicale doit être vérifiée.
- En d'autres termes, **il s'agit toujours de médecins** et ils doivent être traités comme tels.
- **Polymorphisme**

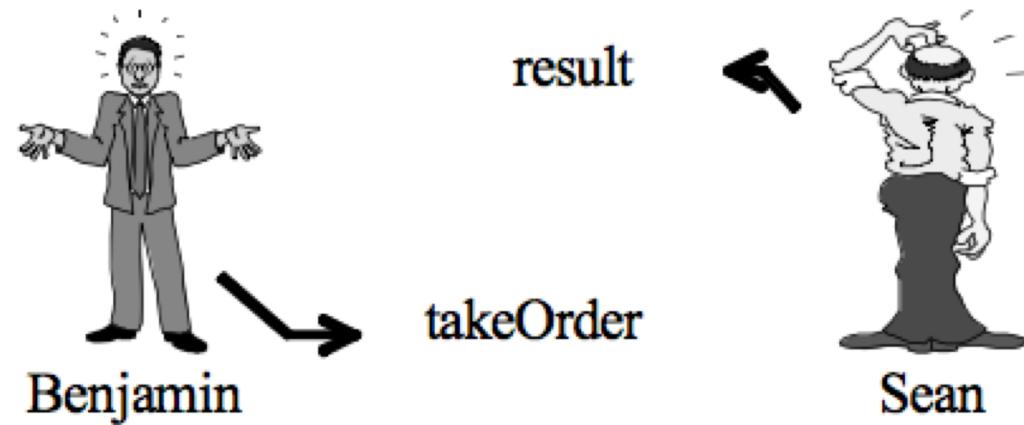
Programmation orientée objet

- Une "classe" est une conception logicielle qui décrit les propriétés générales d'un objet que le logiciel modélise.
- Des "objets" individuels sont créés à partir de la conception de la classe pour chaque objet réel.



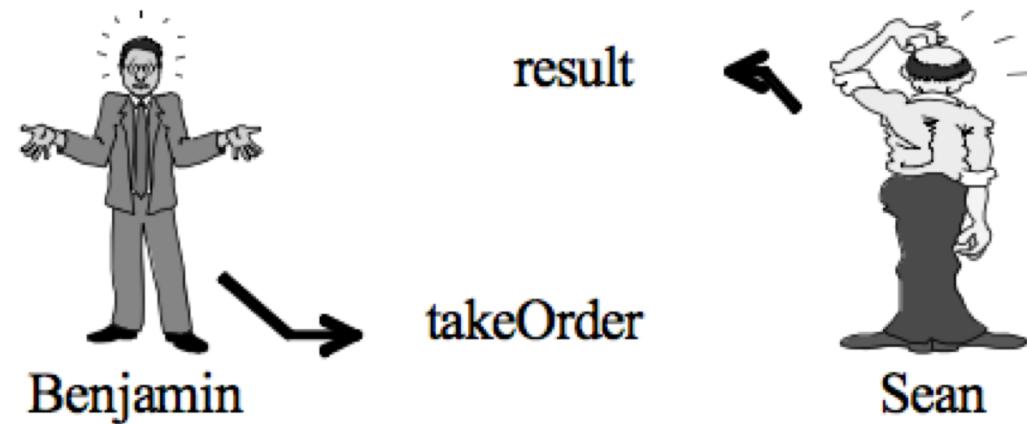
Classes et Objets

Considérons une situation réelle. Il y a deux personnes, Benjamin et sa femme, Barbie. Ils sont clients de HomeCare, une entreprise qui vend des meubles de luxe. HomeCare vend une variété de canapés. Chaque ensemble de canapés est étiqueté avec un numéro d'identification et une étiquette de prix. Après avoir regardé les canapés pendant une heure, Benjamin et Barbie décident d'acheter un ensemble 5 places en cuir vert. Ils s'adressent à Sean, un vendeur, pour passer leur commande.



Classes et Objets

En faisant connaître sa demande à Sean, Benjamin lui envoie le message suivant : "Je voudrais acheter cet ensemble 5 places en cuir vert. Pouvez-vous me l'envoyer d'ici mercredi prochain ?". Le message que Benjamin a envoyé à Sean est un message **takeOrder**. Il contient des informations telles que le type de canapé (un ensemble en cuir vert, 5 places) et la date de livraison (mercredi prochain). Ces informations sont connues comme les paramètres du message **takeOrder**. En réponse au message de Benjamin, Sean répond à Benjamin en lui retournant le résultat de sa demande.



Classes et Objets

Sean a pu répondre au message **takeOrder** de Benjamin car il l'a compris et avait les moyens de traiter la demande de Benjamin.

Bien que Sean ait su comment satisfaire la demande de Benjamin, ce dernier ne l'a pas fait. En fait, la plupart du temps, **les clients ne savent pas comment un vendeur a satisfait leurs commandes**.

Tout ce qu'ils obtiennent des vendeurs, ce sont des réponses telles que : "Je suis désolé, madame, nous ne pouvons pas satisfaire votre demande car le canapé que vous voulez a été vendu" ou "Monsieur, votre demande a été satisfaite. Nous livrerons la marchandise mercredi entre 10h et 11h à l'adresse indiquée. Nous vous remercions pour votre commande."

Classes et Objets

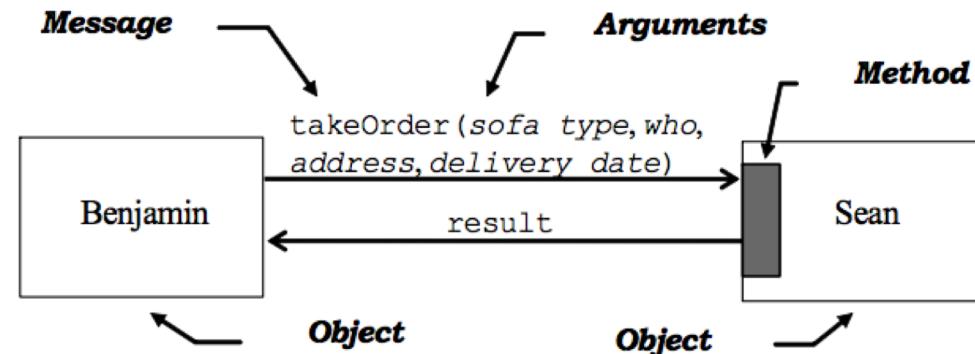
Sean, en tant que vendeur de HomeCare, a une responsabilité envers Benjamin. Il maintient sa responsabilité en appliquant un ensemble d'opérations :

1. Il détermine si le stock est suffisant pour satisfaire la demande de Benjamin.
2. Il détermine si la date demandée pour la livraison est une date convenable.
3. Il donne l'ordre au personnel de l'entrepôt de livrer les marchandises à l'adresse de Benjamin à la date demandée, si les conditions ci-dessus sont remplies.
4. Enfin, il informe Benjamin du résultat de sa demande.

Classes et Objets

Les interactions entre Benjamin et Sean dans la situation réelle ci-dessus peuvent être représentées en termes de programmation orientée objet. Par exemple, Benjamin et Sean sont des objets qui interagissent en envoyant des messages. Benjamin est donc un objet émetteur de messages, tandis que Sean est un objet récepteur de messages.

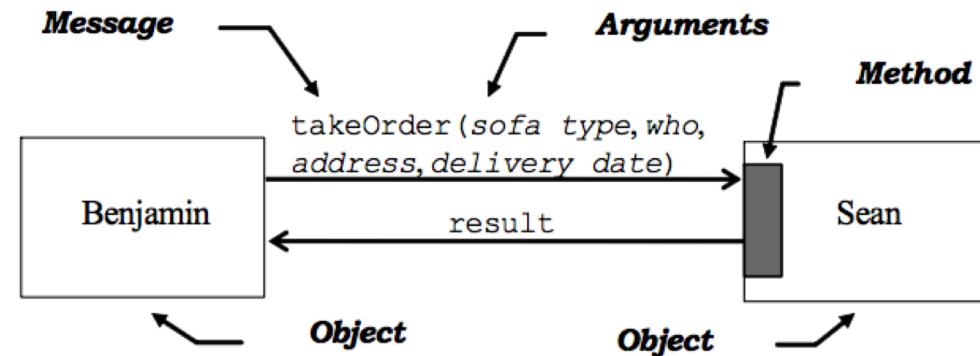
Alternativement, nous pouvons étiqueter Benjamin comme un émetteur et Sean comme un récepteur. La demande `takeOrder` de Benjamin à Sean est un exemple de message. Elle peut être accompagnée d'informations supplémentaires, appelées paramètres (ou arguments) du message.



Classes et Objets

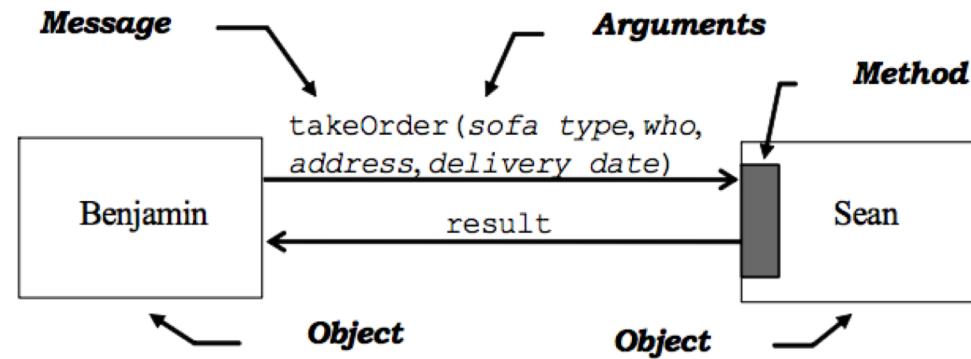
Le fait que Sean ait répondu au message de Benjamin indique que le message est un message valide. Chaque message valide correspond à une méthode que Sean utilise pour remplir sa responsabilité envers Benjamin.

Un message non valide, en revanche, est un message auquel le récepteur n'a pas la capacité de répondre, c'est-à-dire que le récepteur n'a pas de méthode correspondante au message.



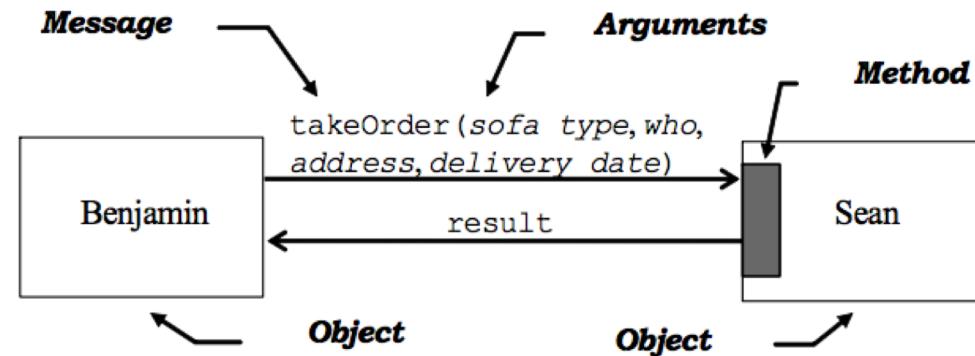
Classes et Objets

- Par exemple, si Benjamin avait demandé une remise sur le prix, sa demande aurait été rejetée car Sean, étant un vendeur, n'aurait pas la capacité (ou une méthode correspondante) de répondre au message.
- Une méthode contient un certain nombre d'opérations détaillant comment Sean doit satisfaire la demande que Benjamin lui a adressée par le biais de sa requête.



Classes et Objets

Même si Benjamin peut **savoir ce que Sean peut faire** par le biais de ses méthodes, il ne peut pas savoir **comment Sean les fait**. Il s'agit d'un principe important de la programmation orientée objet connu sous le nom de **dissimulation** d'informations : l'expéditeur d'un message ne sait pas comment un récepteur va satisfaire la demande contenue dans le message.



Classes et Objets

Benjamin as an Object

Attributes:

```
name = "Benjamin"  
address = "1, Robinson Road"  
budget = "2000"
```

Methods:

```
purchase() {send a purchase request to a salesperson}  
getBudget() {return budget}
```

Bernie as an Object

Attributes:

```
name = "Bernie"  
address = "18, Sophia Road"  
budget = "1000"
```

Methods:

```
purchase() {send a purchase request to a salesperson}  
getBudget() {return budget}
```

Bernie est un autre client de HomeCare. En tant que clients de HomeCare, Benjamin et Bernie partagent certaines informations similaires. Par exemple, ils ont tous deux **un nom, une adresse et un budget - des informations pertinentes pour décrire les clients.**

Programmation orientée objet

Benjamin as an Object

Attributes:

```
name = "Benjamin"  
address = "1, Robinson Road"  
budget = "2000"
```

Methods:

```
purchase() {send a purchase request to a salesperson}  
getBudget() {return budget}
```

Bernie as an Object

Attributes:

```
name = "Bernie"  
address = "18, Sophia Road"  
budget = "1000"
```

Methods:

```
purchase() {send a purchase request to a salesperson}  
getBudget() {return budget}
```

- Ces informations sont connues sous le nom **d'attributs d'objet**.
- Collectivement, les valeurs des attributs d'un objet représentent **l'état de l'objet**.

Objets et classes

Benjamin as an Object

Attributes:

```
name = "Benjamin"  
address = "1, Robinson Road"  
budget = "2000"
```

Methods:

```
purchase() {send a purchase request to a salesperson}  
getBudget() {return budget}
```

Bernie as an Object

Attributes:

```
name = "Bernie"  
address = "18, Sophia Road"  
budget = "1000"
```

Methods:

```
purchase() {send a purchase request to a salesperson}  
getBudget() {return budget}
```

Outre les attributs, Benjamin et Bernie présentent également certains comportements typiques d'un client. Par exemple, Benjamin et Bernie exécutent une méthode lorsqu'ils effectuent un achat. Appelons cette méthode `purchase()`.

La méthode `purchase()` est composée d'un ensemble d'opérations que Benjamin et Bernie utiliseraient pour envoyer une demande d'achat à un vendeur.

Objets et classes

Pour définir ces objets, on utilise une définition commune appelée classe. Une classe est un modèle de définition permettant de structurer et de créer des objets ayant les mêmes attributs et méthodes. Benjamin et Bernie, étant des clients de HomeCare, peuvent donc être définis par une classe appelée Customer.

```
Class Customer
    Attributes:
        name
        address
        budget
    Methods:
        purchase() {send a purchase request to a salesperson}
        getBudget() {return budget}
```

Objets et classes

Une différence majeure entre les objets et les classes réside dans la manière dont les attributs et les méthodes sont traités dans les objets et les classes.

Une classe est une définition des objets ; les attributs et les méthodes d'une classe **sont donc des déclarations qui ne contiennent pas de valeurs**. En revanche, les objets sont des instances créées d'une classe. Chacun d'eux possède ses propres attributs et méthodes. Les valeurs de l'ensemble des attributs décrivent l'état des objets.

Objets et classes

Examinons maintenant les vendeurs. Les vendeurs ont également des attributs et des méthodes. **Sean et Sara sont deux vendeurs de HomeCare.**

Ils sont donc capables d'un comportement typique d'un vendeur, par exemple, prendre les commandes des clients.

Pour remplir leur rôle de vendeurs dans une transaction d'achat, Sean et Sara exécutent une méthode. Nous appellerons cette méthode `takeOrder()`, et nous représenterons Sean et Sara comme suit :

```
Sean as an Object
Attributes:
    name = "Sean"
Methods:
    takeOrder()      {
        check with warehouse on stock availability
        check with warehouse on delivery schedule
        if ok
            then {instruct warehouse to deliver stock(address, date)
                   return ok}
            else return not ok
    }

Sara as an Object
Attributes:
    name = "Sara"
Methods:
    takeOrder()      {
        check with warehouse on stock availability
        check with warehouse on delivery schedule
        if ok
            then {instruct warehouse to deliver stock(address, date)
                   return ok}
            else return not ok
    }
```

Objets et classes

Étant des vendeurs, Sean et Sara partagent, comme prévu, des attributs et des méthodes similaires. Comme les clients, leur définition peut être décrite par une classe appelée SalesPerson avec la représentation suivante

```
Class SalesPerson
    Attributes:
        name
    Methods:
        takeOrder() {
            check with warehouse on stock availability
            check with warehouse on delivery schedule
            if ok
                then {instruct warehouse to deliver stock(address, date)
                      return ok}
            else return not ok
        }
```

Notez que la définition de la classe SalesPerson est différente de celle de la classe Customer, car les clients et les vendeurs se comportent différemment : les clients passent des commandes et les vendeurs en prennent.

Messages et méthodes

- Les objets communiquent entre eux en envoyant des messages. Un message est un appel de méthode d'un objet émetteur de message à un objet récepteur de message. Un objet émetteur de messages est un émetteur et un objet récepteur de messages est un récepteur.
- Un objet répond à un message en exécutant l'une de ses méthodes. Des informations supplémentaires, appelées arguments, peuvent accompagner un appel de méthode. Une telle paramétrisation permet une plus grande flexibilité dans le passage des messages. L'ensemble des méthodes définit collectivement le comportement dynamique d'un objet. Un objet peut avoir autant de méthodes que nécessaire.

Messages et méthodes



Un message est composé de trois éléments :

- un identifiant d'objet qui indique le destinataire du message,
- un nom de méthode (correspondant à la méthode du destinataire), et
- des arguments (informations supplémentaires nécessaires à l'exécution de la méthode).

Objets et classes

- Dans la programmation orientée objet, les objets sont créés à partir de classes. Les instances des objets Customer sont créées à partir d'une classe Customer et les objets SalesPerson à partir d'une classe SalesPerson.
- Les instances d'objets créées sont des individus avec leur propre état. Pour illustrer cela, prenons l'exemple des compteurs. Un compteur est un dispositif qui tient compte du nombre de fois qu'un événement s'est produit. Il possède deux boutons : un bouton d'initialisation qui remet le compteur à 0, et un bouton d'ajout qui ajoute 1 à son nombre actuel.

First Counter Object

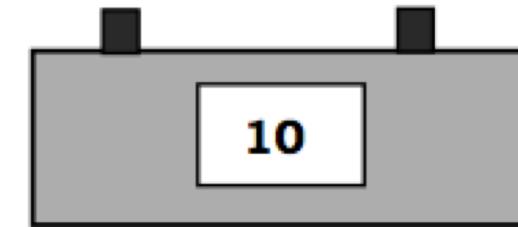
Attributes:

number = 10

Methods:

add()	{number = number + 1}
initialize()	{number = 0}
getNumber()	{return number}

initialize add



Objets et classes

Les trois compteurs partagent la même définition des attributs et des méthodes, et comme dans les exemples précédents, ils peuvent être définis par une classe comme suit :

La classe Counter a :

- un attribut, number ;
- une méthode initialize() qui fait que le compteur remet son numéro à 0.
- une méthode add() qui fait en sorte qu'un compteur ajoute 1 à son nombre ;
- une méthode getNumber() qui renvoie la valeur actuelle de l'attribut number.

Second Counter Object

Attributes:

number = 2

Methods:

add()

initialize()

getNumber()

{number = number + 1}

{number = 0}

{return number}

Third Counter Object

Attributes:

number = 7

Methods:

add()

initialize()

getNumber()

{number = number + 1}

{number = 0}

{return number}

Class Counter

Attributes:

number

Methods:

add()

initialize()

getNumber()

{number = number + 1}

{number = 0}

{return number}

Objets et classes

- Supposons qu'un nouvel objet soit créé à partir de la classe Counter. Bien que le nouvel objet Counter ait la même définition des attributs et des méthodes que les trois compteurs précédents, sa valeur d'attribut peut ne pas être la même que celle des autres compteurs.
- Cela suggère que l'état des compteurs peut être différent les uns des autres.

Objets et classes

```
class Rectangle {  
    Attributes:  
        length  
        width  
    Methods:  
        getLength() { return length }  
        getWidth() { return width }  
        draw() { ... }  
}
```

Objets et classes

- Une personne est une classe ou objet?
- Situation: une personne va conduire sa voiture depuis le point A jusqu'au point B.
- Combien de classes/objets/méthodes on a?

A retenir

- Les objets sont définis par des classes.
- Les objets d'une même classe partagent la même définition des attributs et des méthodes.
- Les objets d'une même classe peuvent ne pas avoir les mêmes valeurs d'attributs.
- Les objets de classes différentes ne partagent pas la même définition des attribut ou de méthodes.
- Les objets créés à partir de la même classe partagent la même définition des attributs et des méthodes, mais leur état peut être différent.
- Une méthode est un ensemble d'opérations exécutées par un objet à la réception d'un message.
- Un message est composé de trois éléments : un identifiant d'objet, un nom de méthode et des arguments.

Get to know JAVA

ABNANE Ibtissam

Objets et classes

- Dans le monde réel => Tout est un **objet**
- Un objet est une entité qui a des:
 - Propriétés permettant d'identifier son État,
 - Méthodes de comportement/fonctionnalité
- **Etat d'un objet:** Données associées à un objet donné à un moment donné

Objets et classes

Classe

- C'est un Plan/Template qui décrit l'état et le comportement que les objets de la classe partagent.
- Une classe peut être utilisée pour créer de nombreux objets.
- Une classe est un modèle pour les objets.

Objets et classes

Object

L'objet est une **instance** d'une classe.

L'objet est une entité du monde réel telle que stylo, ordinateur portable, mobile, lit, clavier, souris, chaise, etc.

L'objet est une **entité physique**.

L'objet est créé plusieurs fois selon l'exigence.

L'objet alloue la mémoire lorsqu'il est créé.

Class

La classe est un **Blueprint/Template** à partir duquel des objets sont créés.

La classe est un groupe d'objets similaires.

La classe est une **entité logique**.

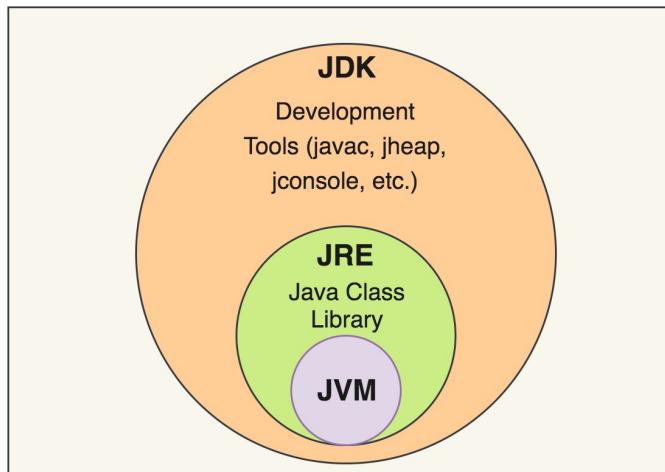
La classe est déclarée une seule fois.

La classe n'alloue pas de mémoire lorsqu'elle est créée.

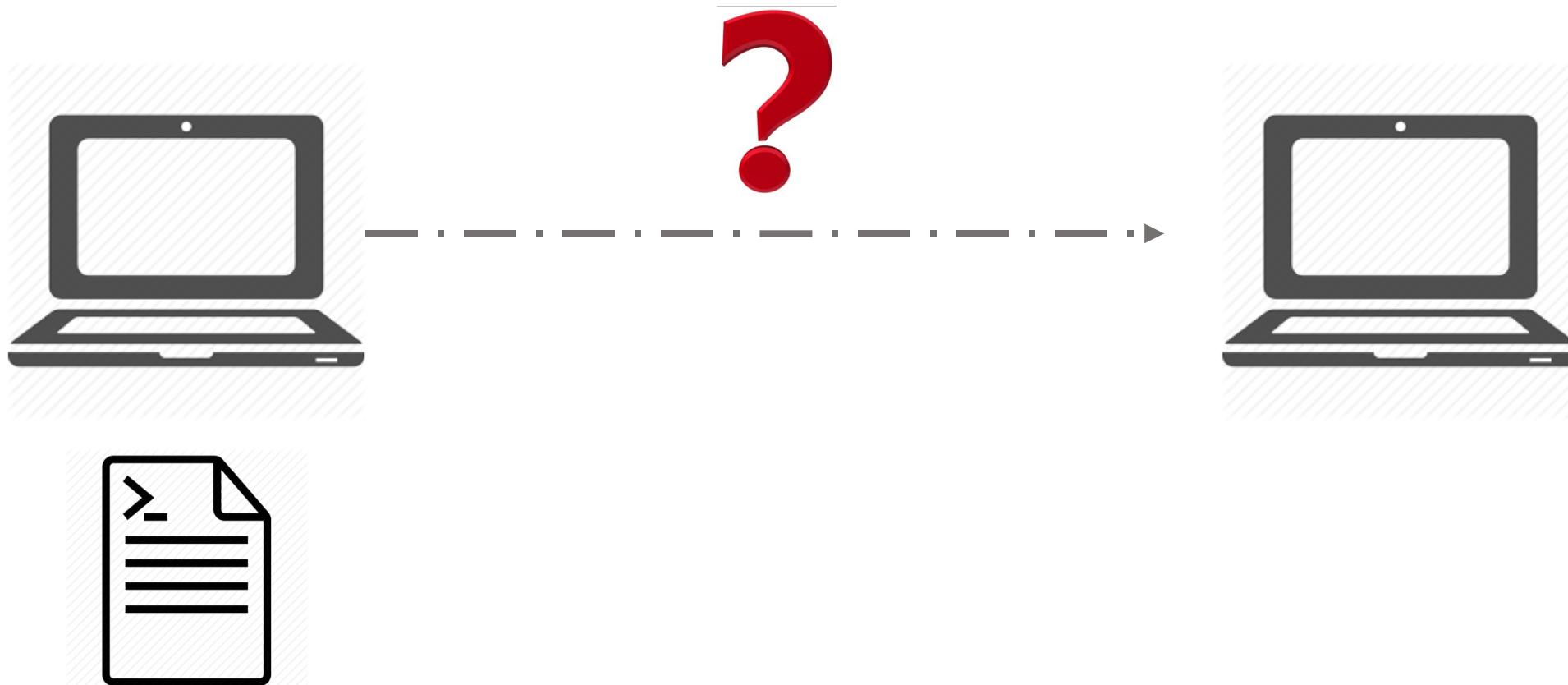
- Java est un langage de programmation **orienté objet, multiplateforme** qui a été lancé par SUN Microsystems en 1995.
- Aujourd'hui, Java est utilisé dans divers secteurs tels que les applications Android, les applications Web Java, les applications de trading, les technologies Big Data, etc.
- Polyvalent destiné à permettre aux programmeurs d'écrire une fois et de s'exécuter n'importe où (WORA)

- le code Java compilé peut s'exécuter sur toutes les plates-formes qui prennent en charge Java sans qu'il soit nécessaire de le recompile
- Les applications Java sont généralement compilées en bytecode qui peut s'exécuter sur toute machine virtuelle Java (JVM), quelle que soit l'architecture informatique sous-jacente.

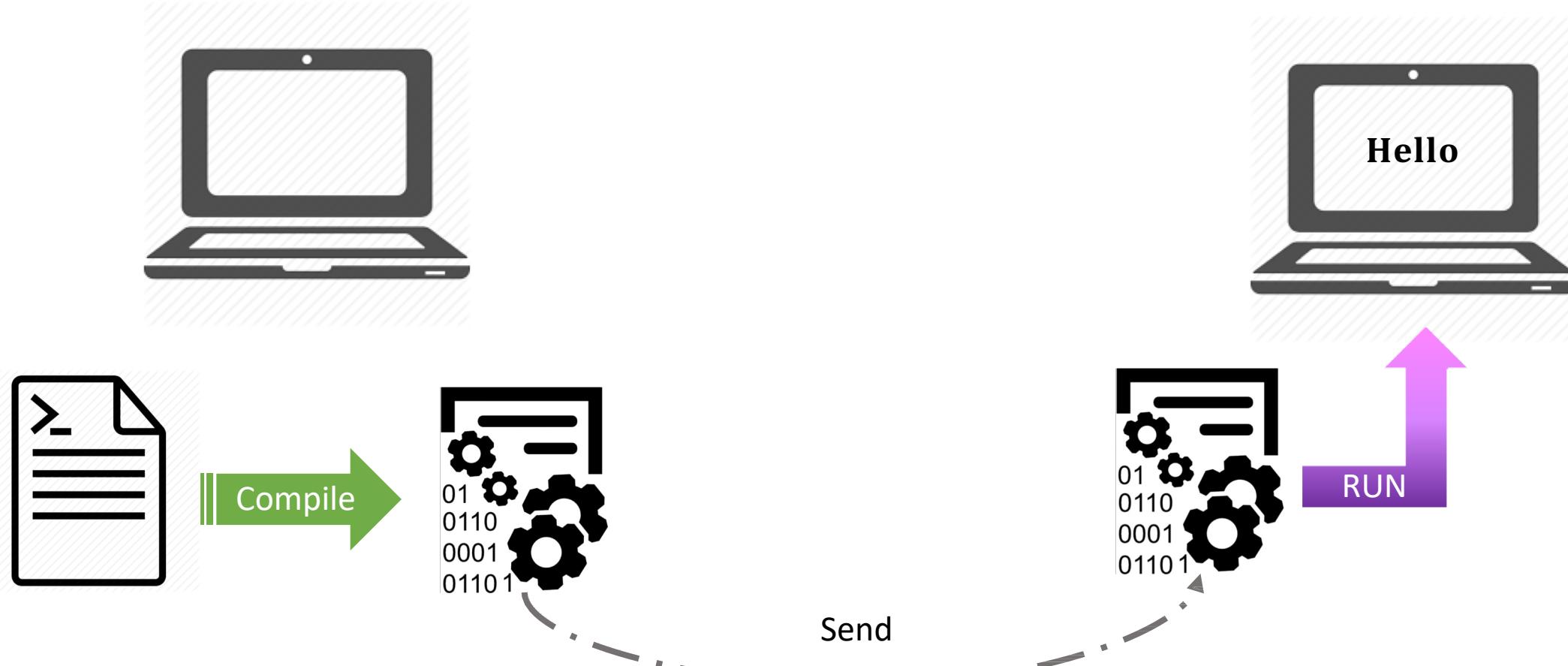
- JRE (Java Runtime Environment) : l'environnement minimum nécessaire pour exécuter une application Java (pas de support pour le développement). Il comprend la JVM (Java Virtual Machine) et les outils de déploiement.
- JDK (Java Development Kit) : l'environnement de développement complet utilisé pour développer et exécuter des applications Java. Il comprend le JRE et les outils de développement.
- **JRE est destiné aux utilisateurs, tandis que JDK est destiné aux programmeurs.**



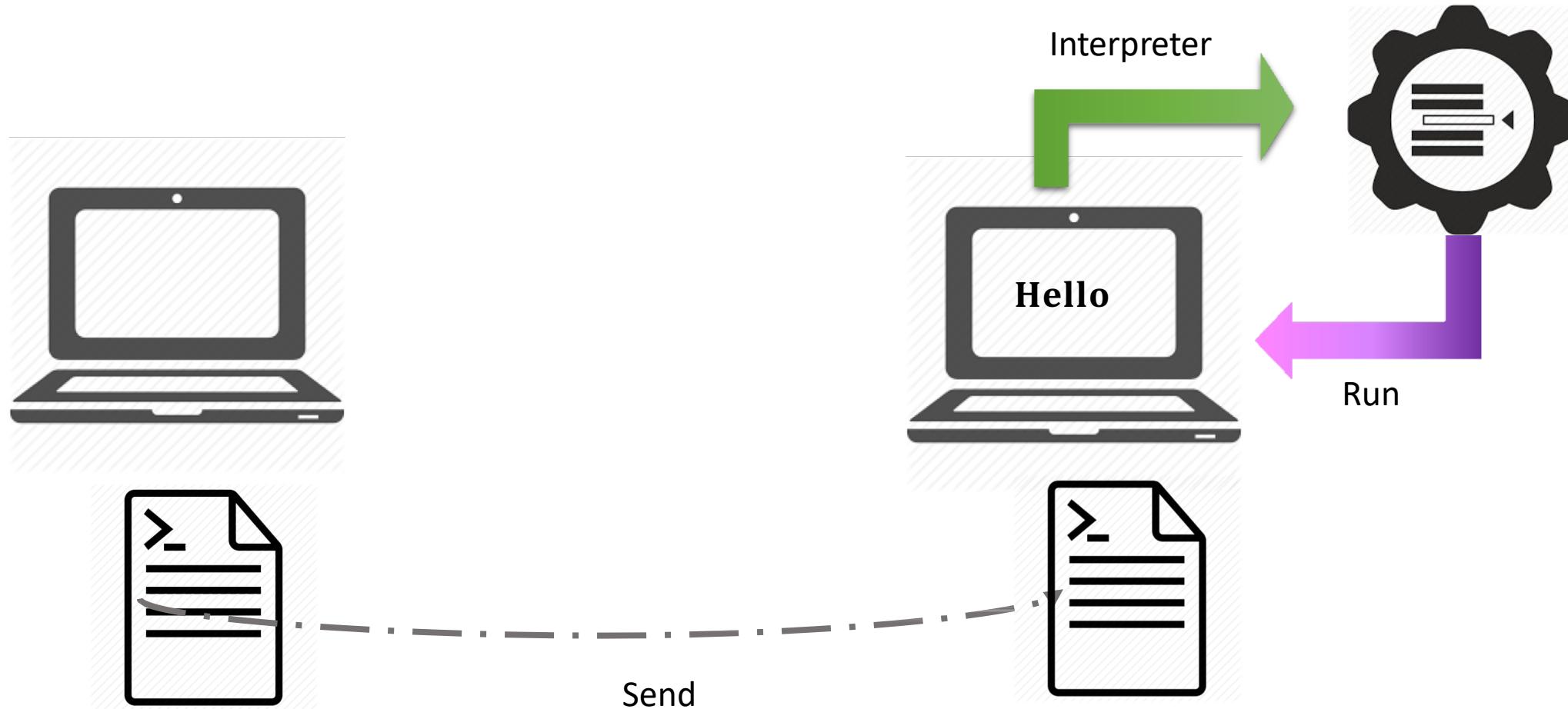
Compiled Vs Interpreted



Compiled



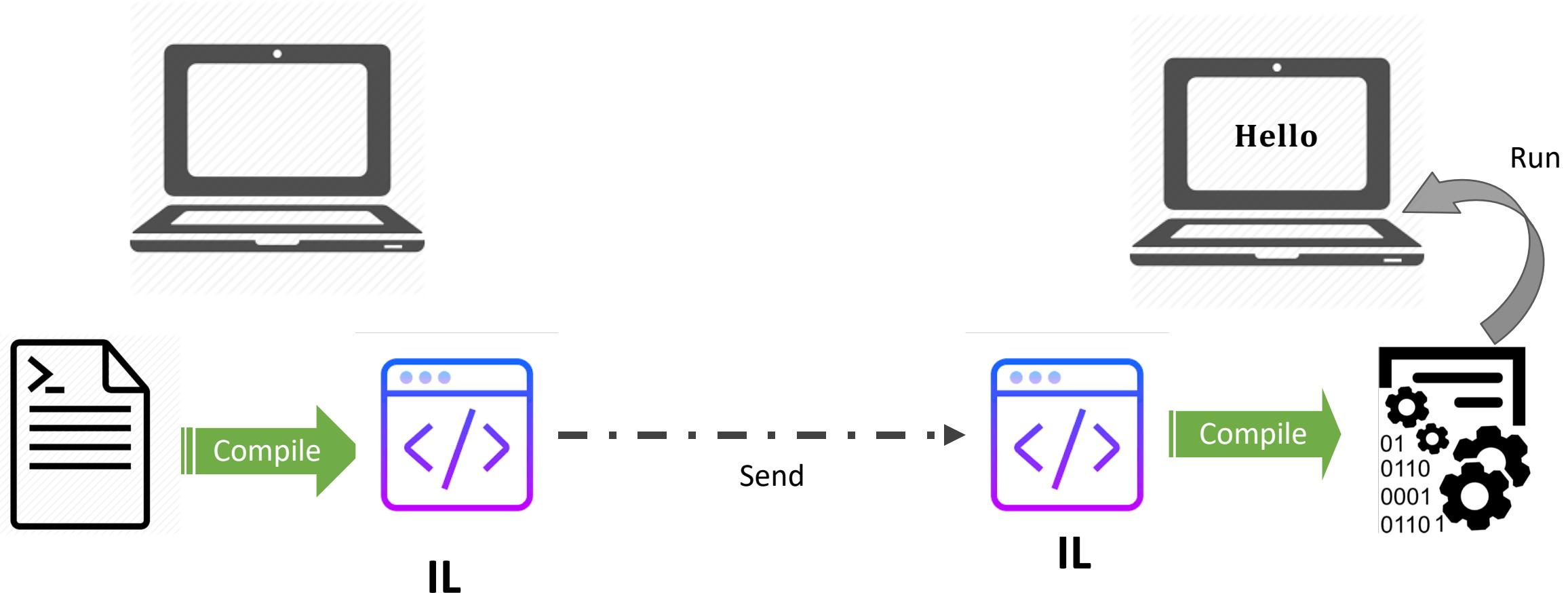
Interpreted



Compiled Vs Interpreted

Compiled		Interpreted	
Pros	Cons	Pros	Cons
Ready to run	Not Cross Platform	Cross Platform	Need an interpreter
Faster	Inflexible	Easy to test	Slower
Private source code	Extra step at modification	Easy to modify	Public source code

Best of Both: Intermediate



- Java est **plateform-independent** => signifie que le code compilé (byte code) de Java peut fonctionner sur tous les systèmes d'exploitation.
- WORA - "write once, run anywhere".
- les codes source Java (fichier .java) sont compilés dans un état **intermédiaire** appelé bytecode (c'est-à-dire un fichier .class) à l'aide du compilateur Java (javac) qui est intégré au JDK.
- Ensuite, la JVM convertit le code d'octet binaire compilé en un langage machine spécifique.
- la JVM est une **spécification** pour un programme logiciel qui exécute le code et fournit l'environnement d'exécution pour ce code.

Le bytecode Java n'est pas un code machine natif, il **ne peut donc pas être exécuté** en tant que tel sur un ordinateur hôte.

Le bytecode sera analysé par un interpréteur spécifique à la plate-forme afin de l'exécuter sur une architecture particulière, comme par exemple Windows, Linux, Mac OS, Sun Solaris, etc.

Ce bytecode est au format hexadécimal avec des lignes opcode-operand et la JVM peut interpréter ces instructions (sans autre recompilation) en langage machine natif qui peut être compris par le système d'exploitation et la plate-forme matérielle sous-jacente.

Java Bytecode - Example

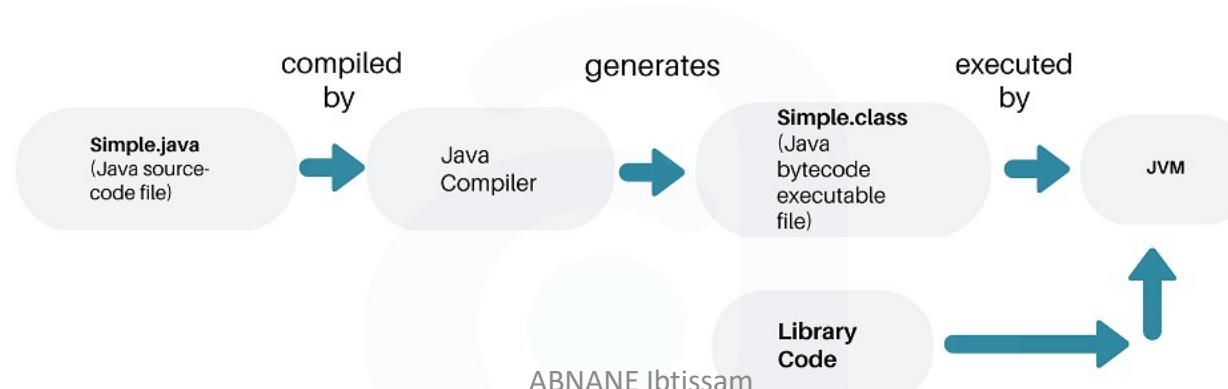
```
Compiled from "Employee.java"
class Employee extends java.lang.Object{
public Employee(java.lang.String,int);
Code:
  0:  aload_0
  1:  invokespecial #1;
       //Method java/lang/Object."<init>":()V
  4:  aload_0
  5:  aload_1
  6:  putfield      #2;
       //Field name:Ljava/lang/String;
  9:  aload_0
 10:  iload_2
 11:  putfield      #3;
       //Field idNumber:I
 14:  aload_0
 15:  aload_1
 16:  iload_2
 17:  invokespecial #4;
       //MethodstoreData:(Ljava/lang/String;I)V
 20:  return

public java.lang.String employeeName();
Code:
  0:  aload_0
  1:  getfield      #2;
       //Field name:Ljava/lang/String;
  4:  areturn

public int employeeNumber();
Code:
  0:  aload_0
  1:  getfield      #3;
       //Field idNumber:I
  4:  ireturn
}
```

Exécution pas à pas d'un programme Java :

- Lorsqu'un programme est écrit en JAVA, le javac le compile.
- Le résultat du compilateur JAVA est le fichier .class ou le bytecode et non le code natif de la machine (contrairement au compilateur C).
- Le bytecode généré est un code non exécutable et a besoin d'un interpréteur pour s'exécuter sur une machine. Cet interprète est la JVM et donc le bytecode est exécuté par la JVM.
- Et finalement le programme s'exécute pour donner la sortie désirée.



- Par exemple, considérons qu'il existe un fichier Java appelé "Test.java". Pour compiler ce fichier de code source, nous devons utiliser la commande suivante.

javac Test.java

- Ici, lorsque "javac" est appelé dans l'invite de commande, il va lire le code java et le compiler dans le fichier de classe en bytecode. Donc pour exécuter ce code, nous devons utiliser le nom de la classe comme suit avec le mot clé "java".

java Test

Le mot clé java crée une instance JVM

Java, étant un langage de programmation indépendant de la plate-forme, ne fonctionne pas selon le principe de la compilation en une étape.

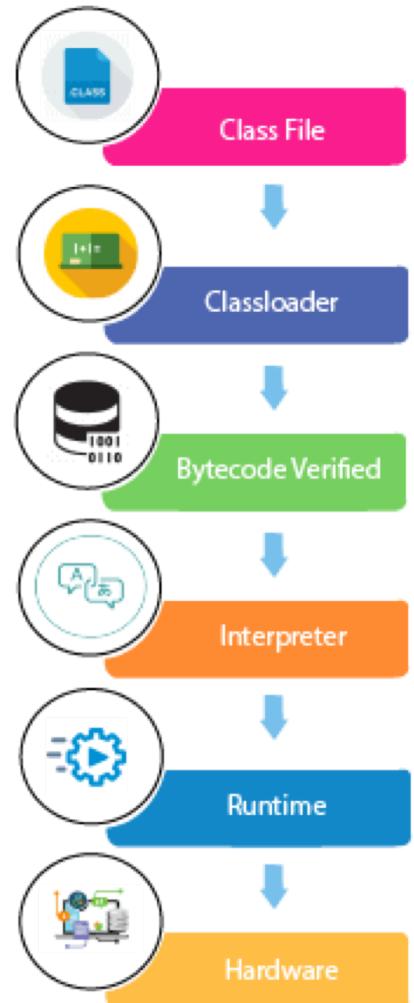
Il implique une exécution en deux étapes, d'abord par le biais d'un **compilateur indépendant** du système d'exploitation, puis dans **une machine virtuelle (JVM)** qui est conçue sur mesure pour chaque système d'exploitation.

Compilation/Compile time

- Tout d'abord, le fichier source '.java' passe par le compilateur, qui encode ensuite le code source dans un codage indépendant de la machine, appelé Bytecode. Le contenu de chaque classe contenue dans le fichier source est stocké dans un fichier '.class' distinct.

Exécution/Runtime

- Les fichiers de classe générés par le compilateur sont indépendants de la machine ou du système d'exploitation, ce qui leur permet d'être exécutés sur n'importe quel système. Pour s'exécuter, le fichier de classe principal (la classe qui contient la méthode main) est transmis à la JVM.



Debugging

Une erreur de programmation est également appelée un bogue, et la procédure permettant de supprimer les erreurs de programmation s'appelle le débogage. Le débogage comporte généralement les trois étapes suivantes :

1. Déetecter la présence d'une erreur.
2. Localiser l'erreur. Cela peut prendre beaucoup de temps pour les gros programmes.
3. Résoudre l'erreur.

Syntax/compilation errors

Une erreur de syntaxe ou de compilation fait référence à une erreur grammaticale dans le programme. Il peut s'agir par exemple d'une erreur de ponctuation ou d'une erreur d'orthographe d'un mot clé. Ces types d'erreurs sont généralement détectés par le compilateur ou l'interpréteur, qui génère un message d'erreur. Prenons l'exemple Java suivant :

BMI= body mass index

```
public void calculate(){  
    BMI = weight / (height*height),  
}
```

L'instruction devrait se terminer par un point-virgule (;) au lieu d'une virgule (,), selon les règles syntaxiques de Java. Par conséquent, une erreur de syntaxe sera générée et affichée. Les erreurs de syntaxe sont généralement faciles à détecter et à résoudre.

Runtime errors

Une erreur d'exécution est une erreur qui se produit pendant l'exécution du programme.

```
public void calculate(){  
    BMI = weight / (height*height);  
}
```

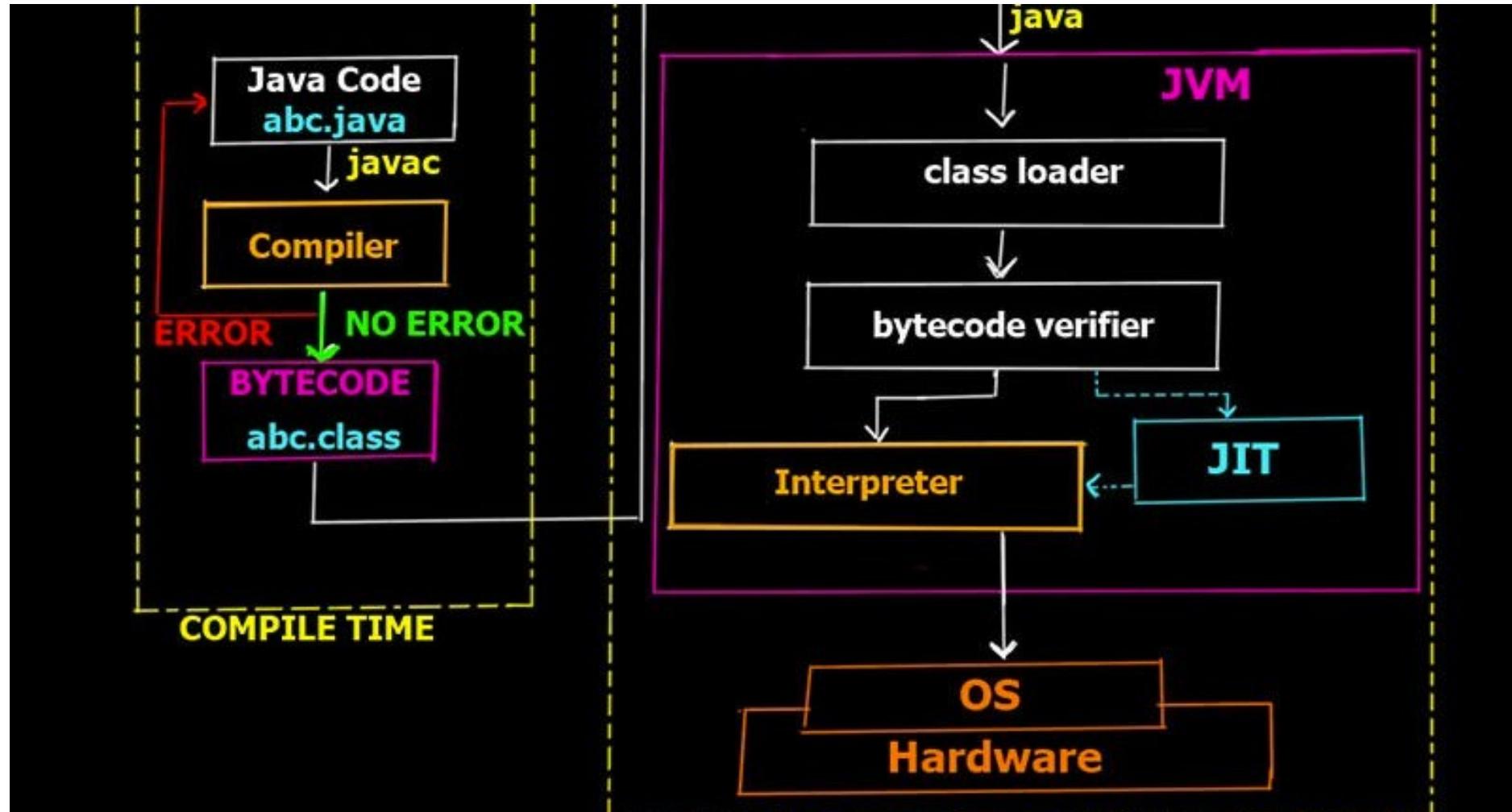
Si l'utilisateur entre une valeur de 0 pour la taille, une division par zéro se produit. Cela crée une erreur d'exécution et provoquera probablement un crash pendant l'exécution. Un autre exemple d'erreur d'exécution est une boucle infinie dans laquelle le programme entre lors de l'exécution. Lors de la conception du programme, il est important de penser aux éventuelles erreurs d'exécution qui pourraient se produire en raison d'une mauvaise saisie de l'utilisateur, qui est à l'origine de la majorité des bogues.

Logic/semantic errors

- Les erreurs logiques ou sémantiques sont les plus difficiles à détecter car le programme fournit une sortie sans générer d'erreur. Cependant, la sortie qui est donnée est incorrecte en raison d'une formule mal programmée.

```
public void calculate(){  
    BMI = (weight*weight) / height;  
}
```

- Cette routine est clairement erronée car elle calcule l'IMC comme (poids*poids)/taille au lieu de poids/(taille*taille). Ces erreurs ne peuvent pas être détectées par les compilateurs ou les interpréteurs.



Retenir: Java est indépendant de la plate-forme mais la JVM dépend de la plate-forme

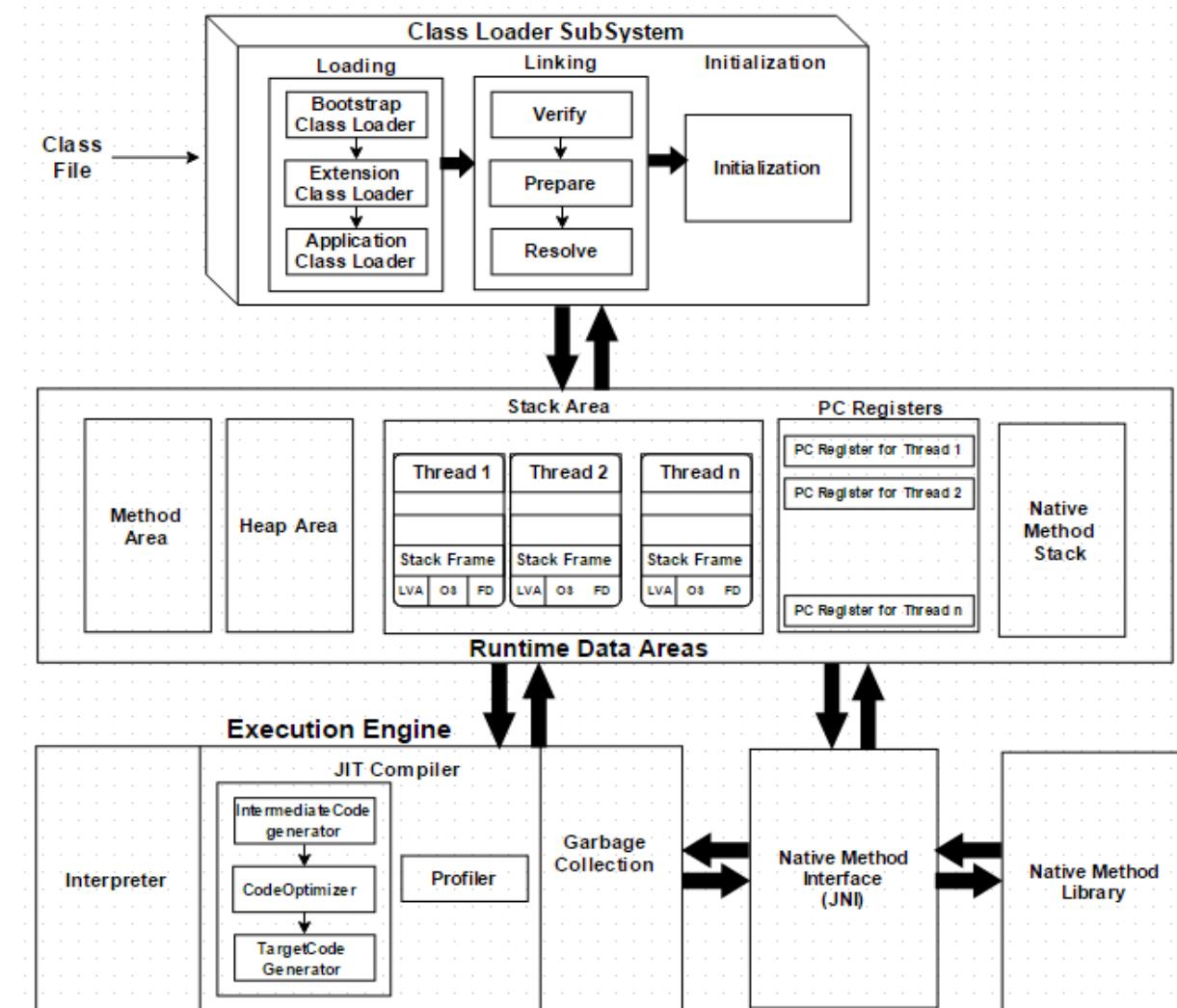
En Java, le point principal est que la JVM dépend du système d'exploitation - ainsi, si vous utilisez Mac OS X, vous aurez une JVM différente de celle que vous utilisez sous Windows ou un autre système d'exploitation.

En essayant de télécharger la JVM pour votre machine particulière, vous obtiendrez une liste de JVM correspondant à différents systèmes d'exploitation, et vous choisirez évidemment la JVM ciblée pour le système d'exploitation que vous exécutez. Nous pouvons donc conclure que la JVM dépend de la plate-forme et que c'est la raison pour laquelle Java est capable de devenir "indépendant de la plate-forme".

- Des interpréteurs ont été développés pour diverses plates-formes. Ils sont tous des implémentations de la machine virtuelle Java (JVM). Le bytecode peut alors être considéré comme du code machine pour la JVM.
- La JVM est essentiellement une unité centrale virtuelle dotée de ses propres codes d'opération. Quelle que soit la plate-forme sur laquelle vous vous trouvez, le bytecode est exactement le même. **La JVM est donc un processus distinct qui fonctionne au-dessus d'un processeur natif.**

- le bytecode agit comme un état intermédiaire indépendant de la plate-forme qui est portable entre toutes les JVM indépendamment du système d'exploitation et de l'architecture matérielle sous-jacents.
- Cependant, comme les JVM sont développées pour fonctionner et communiquer avec la structure matérielle et le système d'exploitation sous-jacents, nous devons sélectionner la version de JVM appropriée pour notre version de système d'exploitation (Windows, Linux, Mac) et l'architecture du processeur.
- La JVM n'est qu'une spécification, et sa mise en œuvre varie d'un fournisseur à l'autre. **Pour l'instant, comprenons l'architecture communément acceptée de la JVM telle que définie dans la spécification.**

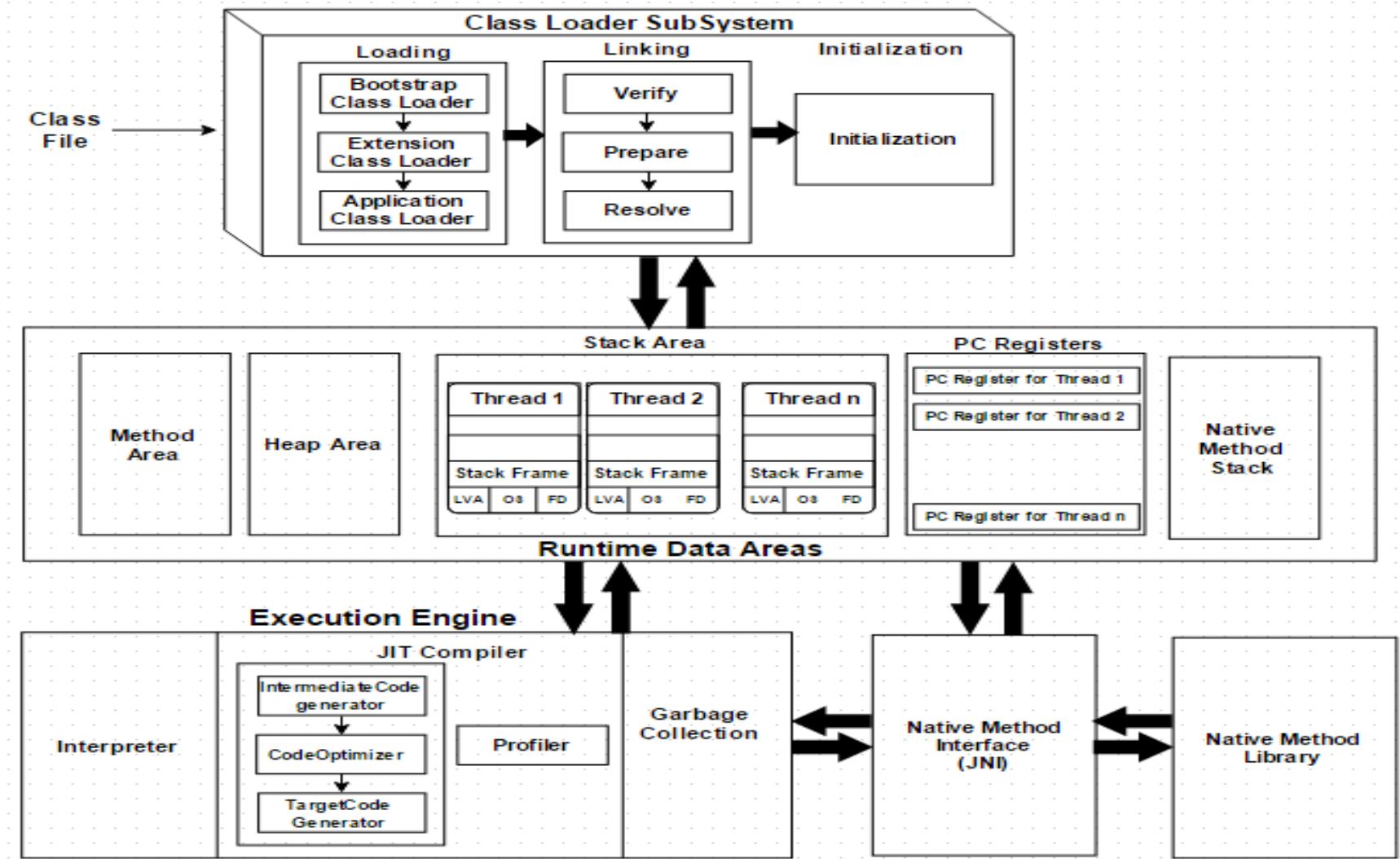
JVM architecture



Garbage collector

- Il vérifie dans la zone du tas s'il y a des objets non référencés et détruit ces objets pour récupérer la mémoire. Il fait ainsi de la place pour les nouveaux objets. Il fonctionne en arrière-plan et rend la mémoire de Java efficace. Ce processus comporte deux phases,
- **Marquer** - Dans cette zone, le Garbage Collector identifie les objets non désirés dans la zone du tas.
- **Balayage** - Dans cette phase, le Garbage Collector supprime les objets de la marque.
- Ce processus est effectué par la JVM à intervalles réguliers et peut également être déclenché en appelant la méthode System.gc().

JVM architecture

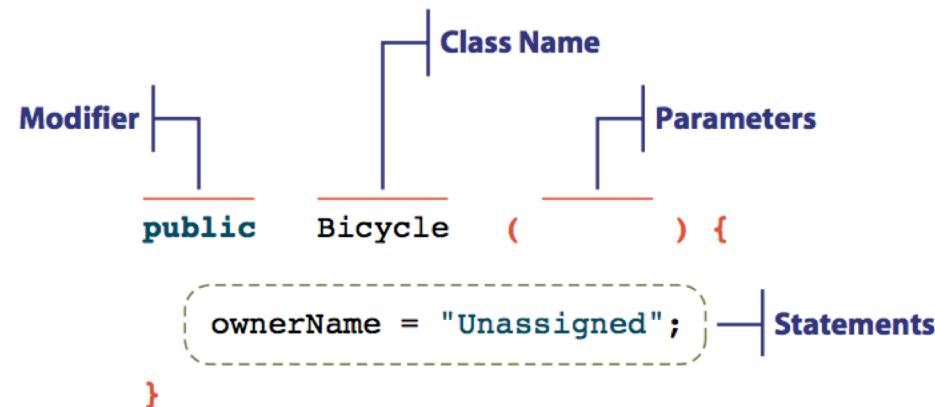


Constructeur

- Un constructeur est une méthode spéciale qui est exécutée lorsqu'une nouvelle instance de la classe est créée, c'est-à-dire lorsque l'opérateur new est appelé. Voici le constructeur de la classe Bicyclette :

```
public Bicycle() {
    ownerName = "Unassigned";
}
```

```
public <class name> (<parameters>) {
    <statements>
}
```

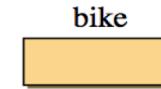


Constructeur

L'objectif du constructeur Bicycle est d'initialiser le membre de données à une valeur qui reflète l'état auquel le nom réel n'est pas encore affecté.

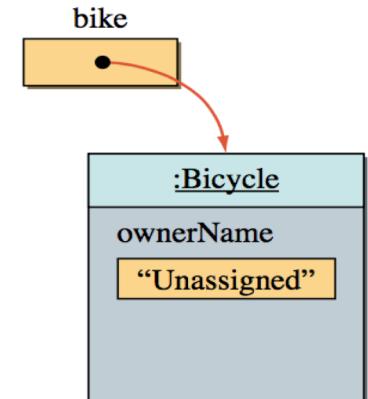
Puisqu'un constructeur est exécuté lorsqu'une nouvelle instance est créée, c'est l'endroit le plus logique pour initialiser les membres de données et effectuer toute autre tâche d'initialisation.

```
    ➔ Bicycle bike;
```



```
Bicycle bike;
```

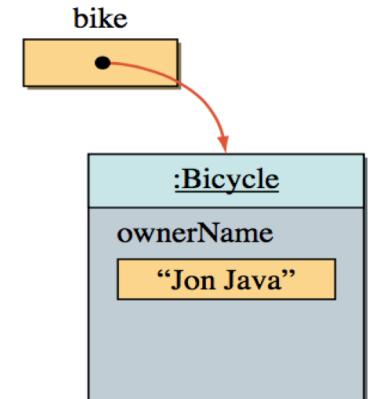
```
    ➔ bike = new Bicycle( );
```



```
Bicycle bike;
```

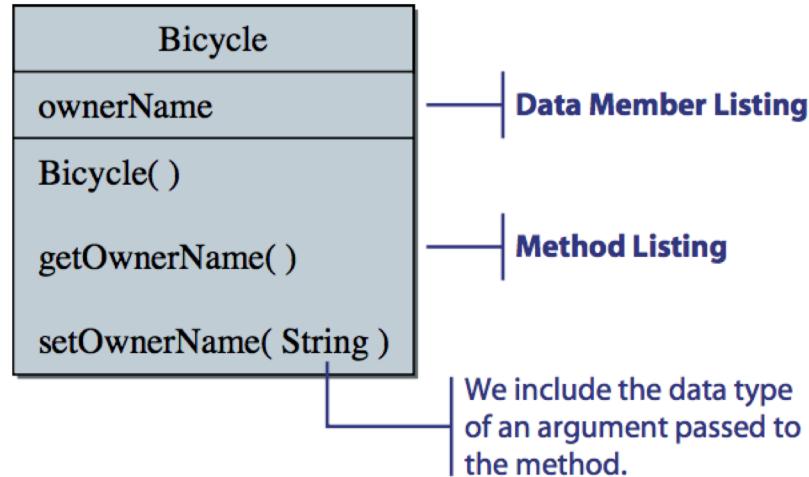
```
    ➔ bike = new Bicycle( );
```

```
    ➔ bike.setOwnerName("Jon Java");
```



Constructeur

- Il est fortement recommandé d'inclure des constructeurs aux classes.
- Cependant, il n'est pas obligatoire de définir explicitement un constructeur dans une classe. Si aucun constructeur n'est défini pour une classe, le compilateur Java inclura automatiquement un constructeur par défaut.
- Un constructeur par défaut est un constructeur qui n'accepte aucun argument et dont le corps ne contient aucune déclaration.
- Nous devons toujours définir notre propre constructeur afin de pouvoir initialiser correctement les membres de données et effectuer toute autre tâche d'initialisation.
- Cela permet de s'assurer qu'un objet est créé dans un état valide (par exemple, si le solde d'un compte est supérieur au minimum).



Structure de la classe: Nous listons d'abord les membres de données, puis le constructeur, et enfin les méthodes. Dans chaque groupe, les éléments sont classés par ordre alphabétique. La convention de regroupement et d'ordonnancement des éléments au sein d'un groupe est destinée à nous faciliter la tâche. Le compilateur Java ne se soucie pas de la façon dont nous listons les membres de données et les méthodes.

Rappel

La programmation orientée objet est un paradigme de programmation où les concepts du programme sont représentés par des "objets".

Chaque objet est une instance d'une classe, qui peut être considérée comme un "plan" ou un modèle des caractéristiques de l'objet.

Contrairement à la programmation procédurale, ces caractéristiques comprennent des données - attributs ou variables décrivant l'état de l'objet - et des comportements - méthodes ou procédures décrivant les actions qu'un objet peut effectuer.

Encapsulation

- La déclaration de classe commence par le mot réservé **class** suivi du nom.
- Tout identifiant valide qui n'est pas un mot réservé peut être utilisé comme nom de classe.

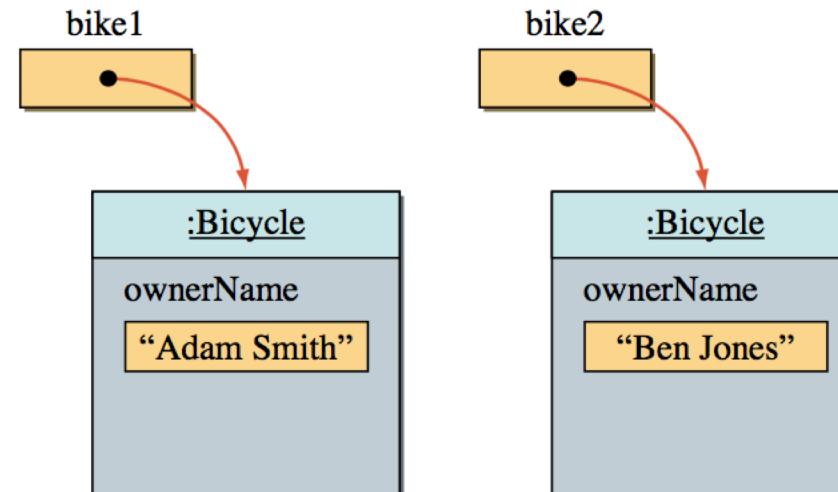
Quels seraient les membres de données des objets Bicycle ?

- Nous devons connaître le nom du propriétaire de chaque objet Bicycle. Nous allons donc définir un membre de données `ownerName` pour stocker le nom du propriétaire. Les membres de données d'une classe sont déclarés dans la déclaration de la classe.

```
class Bicycle {  
    private String ownerName;  
    //definitions for the constructor,  
    //getOwnerName, and setOwnerName methods come here  
}
```

Encapsulation

- Le membre de données `ownerName` est **une variable d'instance**
- Une variable d'instance est le membre de données associé à une instance individuelle et dont la valeur peut changer dans le temps.
- En d'autres termes, chaque instance de la classe aura sa propre copie. Après la création des deux objets `Bicycle` et l'attribution de leurs noms respectifs par le programme, nous avons l'état de mémoire suivant :



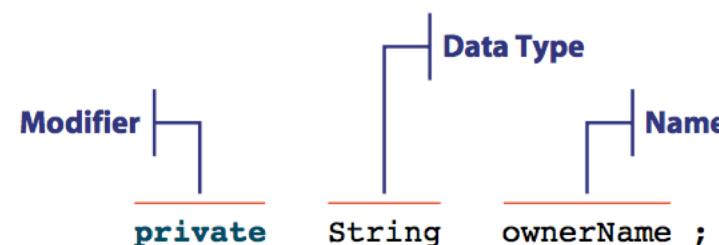
Encapsulation

La syntaxe de la déclaration de l'élément de données est la suivante

<modifier-list> <data type> <name> ;

- <modifier-list> désigne différentes caractéristiques du membre de données,
- <data type> le nom de la classe ou le type de données primitif, et
- <name> le nom du membre de données.

Voici comment la syntaxe générale correspond à la déclaration réelle :



Encapsulation

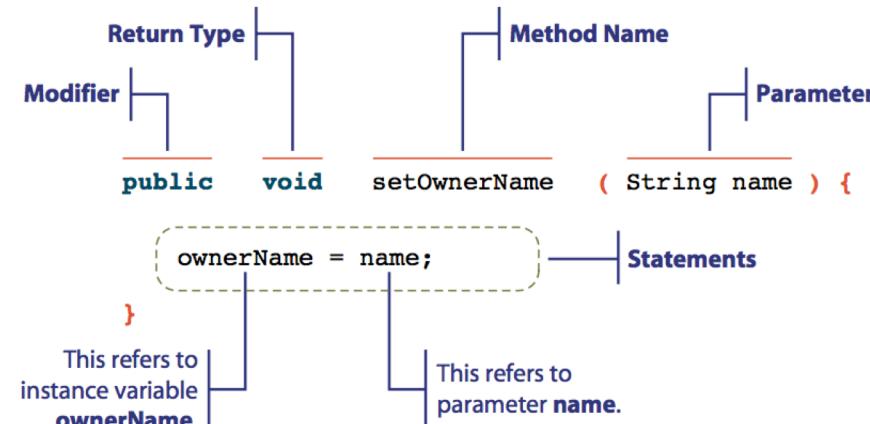
- Dans cet exemple, l'élément de données a un modificateur appelé private.
- Ce modificateur est appelé modificateur d'accessibilité, ou modificateur de visibilité, et il limite qui peut avoir un accès direct au membre de données.
- Si le modificateur est private, alors seules les méthodes définies dans la classe peuvent y accéder directement



Encapsulation

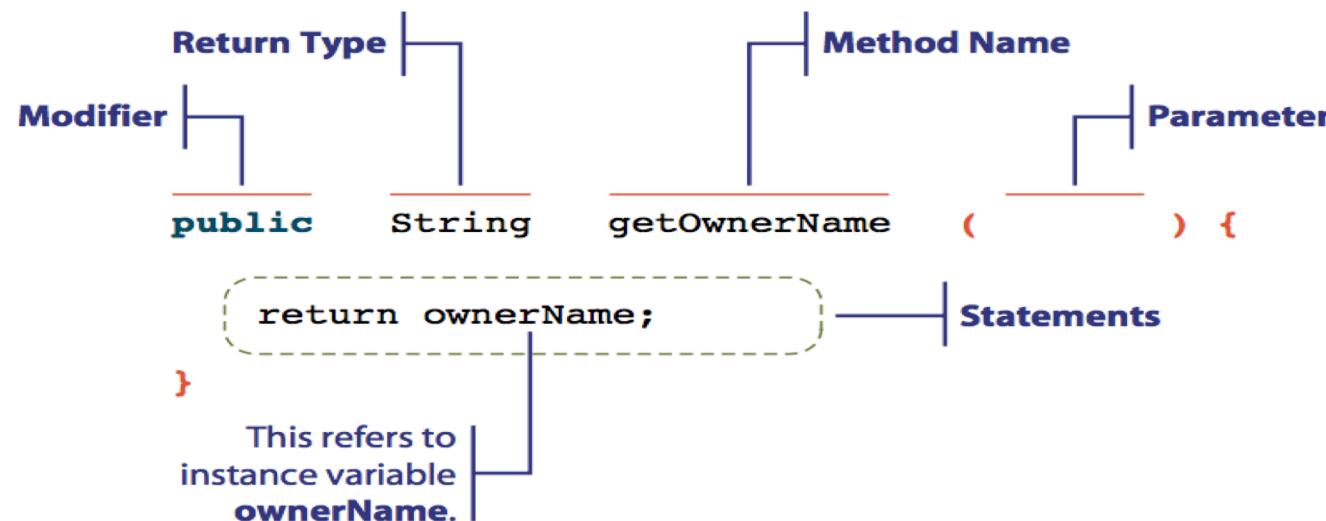
- Les méthodes peuvent ou non renvoyer une valeur.
- Une méthode qui ne renvoie pas de valeur, comme la méthode setOwnerName, est déclarée comme nulle. On l'appelle une méthode void.
- Le modificateur d'accessibilité de la méthode setOwnerName est déclaré public. Cela signifie que le programme qui utilise la classe Bicycle peut accéder à cette méthode, ou l'appeler. Il est possible (et utile) de déclarer une méthode comme privée. Si une méthode est déclarée comme privée, elle ne peut pas être appelée par le programme qui utilise la classe. Elle ne peut être appelée qu'à partir des autres méthodes de la même classe.

```
public void setOwnerName(String name) {
    ownerName = name;
}
```



Encapsulation

- **getOwnerName()**: Il s'agit d'une méthode de retour de valeur. Lorsque cette méthode est appelée, elle renvoie une valeur à l'appelant.
- Renvoie une chaîne de caractères - la valeur de la variable d'instance `ownerName` - et son type de retour est donc déclaré



```
public String getOwnerName( ) {
    return ownerName;
}
```

Encapsulation

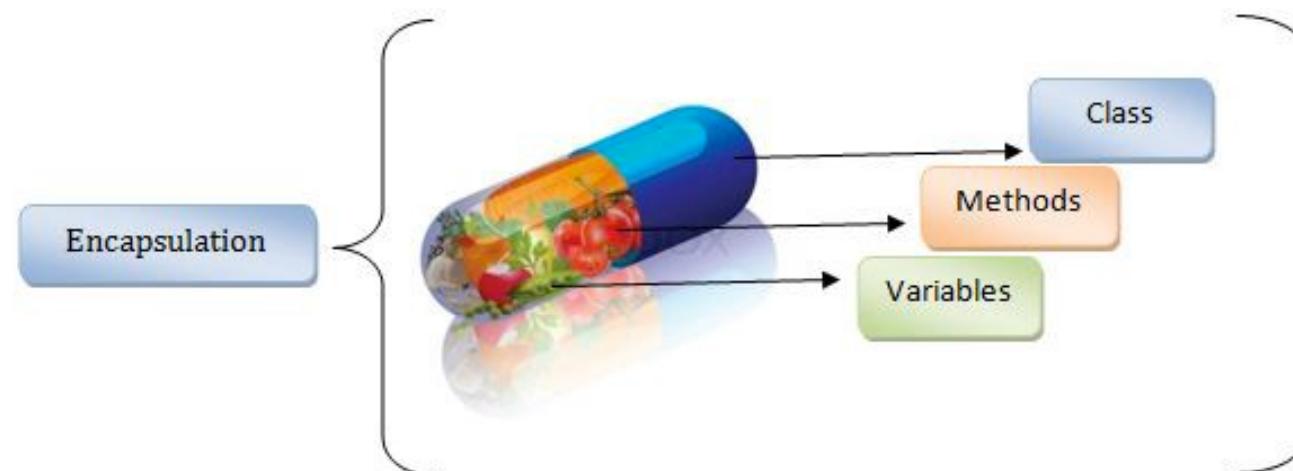
- Une méthode qui renvoie des informations sur un objet (par exemple, qui est le propriétaire d'un vélo) est appelée un accesseur. La méthode `getOwnerName` est un accesseur.
- L'inverse d'un accesseur qui définit une propriété d'un objet est appelé un mutateur. La méthode `setOwnerName` est un mutateur. Les accesseurs et les mutateurs sont communément appelés méthodes **get** et **set**, respectivement.

Encapsulation

- L'une des plus grandes sources d'erreurs dans les programmes est lorsque certaines parties du programme interfèrent avec d'autres parties.
- En effet, il est facile de voir que, dans cet exemple d'administration de cours, l'ajout de procédures et de données supplémentaires conduira rapidement à ce que l'on appelle du **code spaghetti**, où il devient très complexe de suivre la trace de l'exécution car les données peuvent sauter d'une partie à l'autre du programme.
- La programmation orientée objet résout ce problème en encapsulant à la fois **les données et le comportement** dans un objet.

Encapsulation

- Cependant, cela ne suffit pas à garantir des programmes maintenables, car il faut aussi empêcher les données d'un objet d'être directement accessibles par d'autres objets.
- C'est pourquoi la programmation orientée objet met également l'accent sur le concept de **masquage de l'information**, dans lequel les données d'un objet ne sont accessibles par défaut que par des méthodes contenues dans le même objet.



Encapsulation

- Lorsque des éléments de données d'un objet doivent être utilisés par un autre objet, ce dernier doit appeler une méthode publiquement accessible du premier, ce qui revient à demander à "l'objet propriétaire" d'effectuer une modification de ses données.
- En tant que telle, la programmation orientée objet encourage les programmeurs à placer les données là où elles ne sont pas directement accessibles ou modifiables par le reste du système.
- Au lieu de cela, les données sont accessibles par le biais de méthodes, qui peuvent également inclure des contrôles et des sauvegardes pour s'assurer que la modification demandée est autorisée par l'objet propriétaire.

Encapsulation

- Prenons l'exemple d'un robot mobile. Quel type de comportement attendons-nous d'un robot mobile ?
- Des comportements comme avancer, tourner, s'arrêter et changer de vitesse.
- Lorsque nous définissons une classe, disons MobileRobot, nous incluons des méthodes publiques telles que move, turn, stop et changeSpeed. Ces méthodes sont déclarées publiques afin que les programmeurs qui utilisent un objet MobileRobot puissent appeler ces méthodes depuis leurs programmes.
- Nous appelons ces programmeurs des **programmeurs clients** et leurs programmes **des programmes clients**.

Encapsulation

Supposons maintenant que :

- la méthode move accepte un argument entier comme distance à parcourir en mètres.
- Le robot mobile possède trois roues avec un moteur attaché à chacune des roues arrière gauche et droite.
- Le robot n'a pas de mécanisme de direction, de sorte que la rotation s'effectue en faisant tourner les roues arrière gauche et droite à des vitesses différentes. Par exemple, en faisant tourner la roue gauche plus rapidement que la roue droite, le robot effectuera un virage progressif à gauche. Pour avancer, le robot doit envoyer la même quantité d'énergie aux deux moteurs. Pendant que les moteurs tournent, le robot doit constamment surveiller la distance parcourue et arrêter les moteurs lorsque la distance désignée est parcourue.

Encapsulation

- La classe MobileRobot comprend des méthodes telles que rotate pour faire tourner le moteur et readDistance pour lire la distance parcourue.
- Ces méthodes sont déclarées privées car il s'agit de détails internes qui doivent être cachés aux programmeurs clients.
- De notre point de vue de programmeur client, tout ce qui nous importe est que le robot mobile se comporte comme s'il se déplaçait sur la distance souhaitée lorsque nous appelons sa méthode move. Nous ne nous soucions pas de ce qui se passe à l'intérieur. C'est ce qu'on appelle la dissimulation d'informations (information hiding)
- Le nombre de moteurs que possède le robot ou le type de mécanisme utilisé pour le déplacer ne nous intéresse pas. Nous disons que le robot mobile encapsule son fonctionnement interne.

Encapsulation

Pourquoi c'est important?

- Ce mécanisme d'encapsulation permet de modifier plus facilement le code du programme. Supposons, par exemple, que le mécanisme de déplacement d'un robot mobile soit modifié en un moteur unique.
- Les deux roues sont maintenant reliées à un seul essieu, et le moteur fait tourner cet essieu (via des engrenages).
- Le mécanisme interne a changé, mais cela n'affecte pas les programmes clients. L'appel de la méthode move présente toujours le même comportement.

Encapsulation

- Pour mettre en oeuvre ses méthodes (publiques et privées), la classe MobileRobot comprendra nécessairement de nombreux membres de données, tels que la vitesse actuelle, la direction actuelle, les niveaux de puissance des moteurs, etc.
- Ces membres de données sont des queues internes de la classe car il n'est pas du ressort des programmeurs clients de savoir lesquels et combien d'entre eux sont définis dans la classe. En tant que tels, les membres de données sont déclarés comme privés.

A retenir: le comportement des instances est implémenté par des méthodes publiques, tandis que les détails internes qui doivent être cachés aux programmeurs clients sont implémentés par des méthodes privées et des membres de données privés.

Encapsulation

- Le masquage des données et l'encapsulation sont deux concepts de la POO.
- Le masquage des données est le processus qui consiste à protéger les membres de la classe contre tout accès non autorisé.
- L'encapsulation est le processus qui consiste à envelopper les membres des données et les méthodes dans une seule unité.
- C'est la différence entre le masquage des données et l'encapsulation. L'encapsulation est un moyen de réaliser le masquage des données.

Encapsulation

Prenons maintenant un exemple concret pour voir ce qui se passe si un élément qui devrait être un détail interne est déclaré public. Pour illustrer pourquoi déclarer des membres de données publics est considéré comme une mauvaise conception, considérons la classe AccountVer2.

```
class AccountVer2 {  
    public double balance;  
    //the rest is the same  
}
```

Encapsulation

- Si c'était la définition de la classe, nous ne pourrions pas interdire aux programmeurs clients d'écrire du code tel que

```
AccountVer2 myAcct;  
  
myAcct = new AccountVer2("John Smith", 300.00);  
  
myAcct.balance = 670.00;
```

```
class AccountVer2 {  
  
    public double balance;  
  
    //the rest is the same  
}
```

- Cela brise la classe AccountVer2 car le solde peut être modifié directement par les programmeurs clients.
- Les clients auront un accès direct à l'élément de données balance. Ils peuvent modifier sa valeur comme ils le souhaitent.

Encapsulation

- Si la variable d'instance balance est correctement cachée en la déclarant privée, les programmeurs clients ne peuvent pas modifier sa valeur directement.
- Cela préserve l'intégrité de la classe, car les valeurs des membres des données ne sont modifiées que par les méthodes publiques fournies par le concepteur de la classe.
- Les programmes clients ne peuvent pas accéder ou modifier les membres des données.

Retenir: Déclarer les membres de données privés garantit l'intégrité de la classe.

Encapsulation

Si le membre Speed est privé, l'énoncé suivant est-il valide dans le programme du client ?

```
Robot aibo;  
aibo = new Robot();  
double currentSpeed = aibo.speed;
```

Encapsulation

Définissons une autre version de la classe Account (AccountVer3). Cette fois-ci, nous allons facturer des frais fixes à chaque fois qu'une déduction est effectuée. Voici comment la classe est déclarée

```
class AccountVer3 {  
  
    // Data Members  
    private static final double FEE = 0.50;           ← Class constant declaration  
  
    private String ownerName;  
  
    private double balance;  
  
    //Constructor  
    public AccountVer3(String name, double startingBalance) {  
  
        ownerName = name;  
        balance = startingBalance;  
    }  
  
    //Deducts the passed amount from the balance  
    public void deduct(double amt) {  
        balance = balance - amt - FEE;                  ← Fee is charged every time  
    }  
}
```

Encapsulation



L'exemple de programme suivant montre que la taxe de 1,50 \$ est facturée après trois déductions.

```
import java.text.*;  
  
class DeductionWithFee {  
    //This sample program deducts money three times  
    //from the account  
  
    public static void main(String[] args) {  
        DecimalFormat df = new DecimalFormat("0.00");  
        AccountVer3 acct;  
  
        acct = new AccountVer3("Carl Smith", 50.00);  
  
        acct.deduct(10);  
        acct.deduct(10);  
        acct.deduct(10);  
        System.out.println("Owner: " + acct.getOwnerName());  
        System.out.println("Bal  : $"  
                           + df.format(acct.getCurrentBalance()));  
    }  
}
```

Owner: Carl Smith
Bal : \$18.50

Encapsulation

- La constante de classe FEE est déclarée comme suit
`private static final double FEE = 0.50 ;`
- Le modificateur final désigne que l'identifiant FEE est une constante, et le modificateur static désigne qu'il s'agit d'une constante de classe.
- Le mot réservé static est utilisé pour désigner les composants de classe, tels que les variables de classe et les méthodes de classe.
- L'inclusion du mot réservé static dans la déclaration de la méthode main indique qu'il s'agit d'une méthode de classe.

Encapsulation

```
class AccountVer3 {  
    private final double FEE = 0.50;  
    //the rest is the same  
}
```

A large, semi-transparent watermark reading "Bad Version" is overlaid on the code block.

- Cette déclaration n'est pas une erreur, mais elle est inefficace.
- Si FEE est déclarée comme une constante de classe, alors il n'y aura qu'une seule copie pour la classe, et cette copie unique est partagée par toutes les instances de la classe.
- Si FEE est déclaré sans le modificateur statique, alors c'est une constante d'instance. Cela signifie que chaque instance de la classe aura sa propre copie de la même valeur. Par exemple, au lieu d'une copie de la valeur 0,50, il y aura 100 copies de la même valeur 0,50 s'il y a 100 instances de la classe. Ainsi, pour utiliser efficacement une mémoire, lorsque nous déclarons un membre de données comme une constante, il doit être déclaré comme une constante de classe.

Encapsulation

Les membres des données doivent être déclarés privés pour garantir l'intégrité d'une classe.

- Mais il existe une exception. Nous pouvons vouloir déclarer certains types de constantes de classe comme publics.
 1. Une constante est "en lecture seule" de par sa nature, elle n'aura donc pas d'impact négatif si nous la déclarons publique.
 2. Une constante est un moyen propre de faire connaître certaines caractéristiques des instances aux programmes clients.

Encapsulation

- Par exemple, si nous voulons que le montant d'une redevance soit connu du public (ce qui est une bonne idée, car les consommateurs ont besoin de connaître ce type d'information), nous rendons la constante de classe publique comme suit :

```
class AccountVer3 {  
    public static final double FEE = 0.50;  
    ...  
}
```

- Un programme client peut alors accéder directement à cette information comme suit

```
System.out.println("Fee charged per deduction is $ "  
+ AccountVer3.FEE);
```

Encapsulation

- Nous avons souvent besoin d'utiliser des variables temporaires tout en accomplissant une tâche dans une méthode.
- Considérons la méthode deduct de la classe Account :

```
public void deduct(double amt) {  
    balance = balance - amt;  
}
```

- Nous pouvons réécrire la méthode, en utilisant une variable locale, comme suit :

```
public void deduct(double amt) {  
    double newBalance;  
  
    newBalance = balance - amt;  
    balance = newBalance;  
}
```



This is a local variable

Encapsulation

- La variable newBalance est appelée une variable locale. Elles sont déclarées dans la déclaration de la méthode et utilisées à des fins temporaires, comme le stockage des résultats intermédiaires d'un calcul.

```
public void deduct(double amt) {  
    double newBalance;  
  
    newBalance = balance - amt;  
  
    balance = newBalance;  
}
```



This is a local variable

Encapsulation

Cette affectation en deux étapes pour mettre à jour le solde actuel peut ne pas sembler très utile ici, mais envisagez une situation dans laquelle nous devons vérifier certaines conditions avant de modifier réellement la valeur de solde actuel. Par exemple: Interdire l'achat si le solde est inférieur à un solde minimum prédéfini.

```
public void deduct(double amt) {  
    double newBalance;  
  
    newBalance = balance - amt;  
    balance = newBalance;  
}
```

This is a local
variable

Encapsulation

- Alors que les membres de données d'une classe sont accessibles depuis toutes les méthodes d'instance de la classe, les variables locales et les paramètres ne sont accessibles que depuis la méthode dans laquelle ils sont déclarés, et ils ne sont disponibles que pendant l'exécution de la méthode. L'espace mémoire pour les variables locales et les paramètres est alloué respectivement lors de la déclaration et au début de la méthode, et effacé à la sortie de la méthode.
- **Retenir: Les variables locales et les paramètres sont effacés lorsque l'exécution d'une méthode est terminée.**

Encapsulation

- Lorsque vous déclarez une variable locale, assurez-vous que l'identifiant que vous utilisez pour celle-ci n'entre pas en conflit avec les membres de données d'une classe.
- Cette déclaration de classe n'est pas une erreur. Il est acceptable d'utiliser le même identificateur pour une variable locale, mais ce n'est pas conseillé

```
class Sample {  
    private int number;  
    ...  
    public void doSomething() {  
        int number;  
        number = 15;  
    }  
    ...  
}
```

This changes the value
of the local variable, not
the instance variable.

The same identifier is used for both the
local variable and the instance variable.

This

- En Java, le mot-clé **this** est utilisé pour faire référence à l'objet courant à l'intérieur d'une méthode ou d'un constructeur.
- Nous avons créé un objet nommé obj de la classe Main. Nous affichons ensuite la référence à l'objet obj et le mot-clé this de la classe.
- Ici, nous pouvons voir que la référence de obj et this est la même. Cela signifie que this n'est rien d'autre que la référence à l'objet courant.

```
class Main {  
    int instVar;  
  
    Main(int instVar){  
        this.instVar = instVar;  
        System.out.println("this reference = " + this);  
    }  
  
    public static void main(String[] args) {  
        Main obj = new Main(8);  
        System.out.println("object reference = " + obj);  
    }  
}
```

```
this reference = Main@23fc625e  
object reference = Main@23fc625e
```

This

- Il existe plusieurs situations où le mot-clé this est couramment utilisé.
- En Java, il n'est pas permis de déclarer deux ou plusieurs variables ayant le même nom à l'intérieur d'une portée (portée de classe ou portée de méthode). Toutefois, les variables d'instance et les paramètres peuvent porter le même nom. Par exemple,

```
class MyClass {  
    // instance variable  
    int age;  
  
    // parameter  
    MyClass(int age){  
        age = age;  
    }  
}
```

- Dans ce programme, la variable d'instance et le paramètre ont le même nom « age ». Ici, le compilateur Java est confus en raison de l'ambiguïté du nom.

This

- Nous avons passé 8 comme valeur au constructeur. Cependant, nous obtenons 0 en sortie. Ceci est dû au fait que le compilateur Java s'embrouille à cause de l'ambiguïté des noms entre l'instance la variable et le paramètre.

```
class Main {  
  
    int age;  
    Main(int age){  
        age = age;  
    }  
  
    public static void main(String[] args) {  
        Main obj = new Main(8);  
        System.out.println("obj.age = " + obj.age);  
    }  
}
```

```
obj.age = 0
```

This

- Maintenant, nous obtenons le résultat attendu.
- Lorsque le constructeur est appelé, cet objet à l'intérieur du constructeur est remplacé par l'objet obj qui a appelé le constructeur. Par conséquent, la valeur 8 est attribuée à la variable age.

```
class Main {  
  
    int age;  
    Main(int age){  
        this.age = age;  
    }  
  
    public static void main(String[] args) {  
        Main obj = new Main(8);  
        System.out.println("obj.age = " + obj.age);  
    }  
}
```

Une chaîne de caractères est une séquence de caractères qui est traitée comme une valeur unique. Les instances de la classe String sont utilisées pour représenter les chaînes de caractères en Java.

String Length()

- **Les méthodes utilisées pour obtenir des informations sur un objet sont appelées méthodes d'accès.**
- Une méthode d'accès que vous pouvez utiliser avec les chaînes de caractères est la méthode length(), qui renvoie le nombre de caractères contenus dans l'objet chaîne.

String substring (int i)

Return the substring from the i^{th} index character to end.

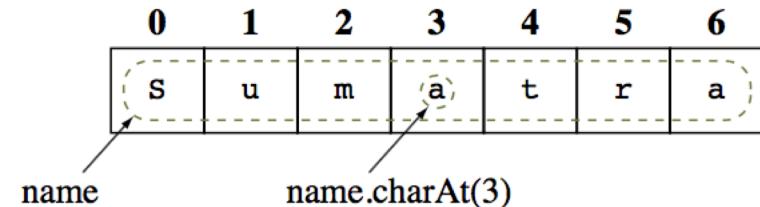
- Disons que nous voulons entrer le nom d'une personne et déterminer le nombre de voyelles qu'il contient. L'idée de base est très simple :
- Il y a deux détails que nous devons connaître avant de pouvoir traduire cela en code réel.
 - 1) nous devons savoir comment faire référence à un caractère individuel dans la chaîne.
 - 2) Deuxièmement, nous devons savoir comment déterminer la taille de la chaîne, c'est-à-dire le nombre de caractères qu'elle contient, afin de pouvoir écrire correctement l'expression booléenne pour arrêter la boucle.

```
for each character ch in the string {
    if (ch is a vowel) {
        increment the counter
    }
}
```

- On accède aux différents caractères d'une chaîne de caractères en appelant la méthode charAt de l'objet String. Par exemple, pour afficher les caractères individuels de la chaîne Sumatra un par un, nous pouvons écrire

```
String name = "Sumatra";
int size = name.length();
for (int i = 0; i < size; i++) {
    System.out.println(name.charAt(i));
}
```

```
String name = "Sumatra";
```



The variable refers to the whole string.

The method returns the character at position 3.

- Strictement parlant, nous devons dire "nom est une variable de type String dont la valeur est une référence à une instance de String". Cependant, il est plus courant de dire "X est une instance de Y" ou "X est un objet Y". Nous abrégeons souvent en disant "X est un Y. »

```
String name1 = "Kona";  
String name2;  
name2 = "Espresso";
```

Raccourci pour

```
String name1 = new String("Kona");  
String name2;  
name2 = new String("Espresso");
```

- Sachez que ce raccourci ne fonctionne que pour la classe String.

```

public static void main (String[] args) {

    Scanner scanner = new Scanner(System.in);
    scanner.useDelimiter(System.getProperty("line.separator"));

    String name;

    int      numberOfCharacters,
            vowelCount = 0;

    char     letter;

    System.out.print("What is your name?");
    name = scanner.next( );

    numberOfCharacters = name.length( );

    for (int i = 0; i < numberOfCharacters; i++) {

        letter = name.charAt(i);

        if (letter == 'a' || letter == 'A' ||
            letter == 'e' || letter == 'E' ||
            letter == 'i' || letter == 'I' ||
            letter == 'o' || letter == 'O' ||
            letter == 'u' || letter == 'U' ) {

            vowelCount++;

        }

    }

    System.out.println(name + ", your name has " +
                       vowelCount + " vowels");
}

```

Version améliorée

```

public static void main (String[] args) {

    Scanner scanner = new Scanner(System.in);
    scanner.useDelimiter(System.getProperty("line.separator"));

    String      name, nameUpper;
    int         numberOfCharacters,
                vowelCount = 0;
    char        letter;

    System.out.print("What is your name?");
    name = scanner.next( );

    numberOfCharacters = name.length( );
    nameUpper = name.toUpperCase( );

    for (int i = 0; i < numberOfCharacters; i++) {

        letter = nameUpper.charAt(i);

        if (letter == 'A' ||
            letter == 'E' ||
            letter == 'I' ||
            letter == 'O' ||
            letter == 'U' ) {

            vowelCount++;

        }

    }

    (name + ", your name has " +
     vowelCount + " vowels");
}

```

Remarquez que **name** est inchangé. Une nouvelle chaîne de caractère **NameUpper** est renvoyée par la méthode **toUpperCase**.

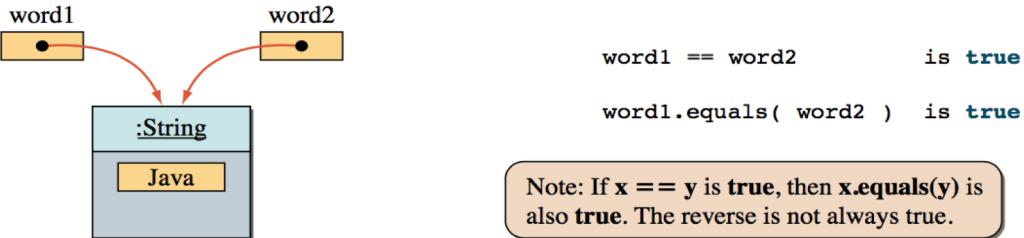
- Comparing strings
- Le test d'égalité `==` est vrai si le contenu des variables est le même. Pour un type de données primitif, les contenus sont les valeurs elles-mêmes, mais pour un type de données de référence, **les contenus sont des adresses**.
- Ainsi, pour un type de données de référence, le test d'égalité est vrai si les deux variables **font référence au même objet**, car elles contiennent toutes deux la même adresse.
- La méthode `equals`, en revanche, est vraie si les objets String auxquels les deux variables font référence **contiennent la même valeur de chaîne**. Pour distinguer les deux types de comparaisons, nous utiliserons le terme test d'équivalence pour la méthode `equals`.

```
String word1, word2;  
...  
if ( word1 == word2 ) ...
```

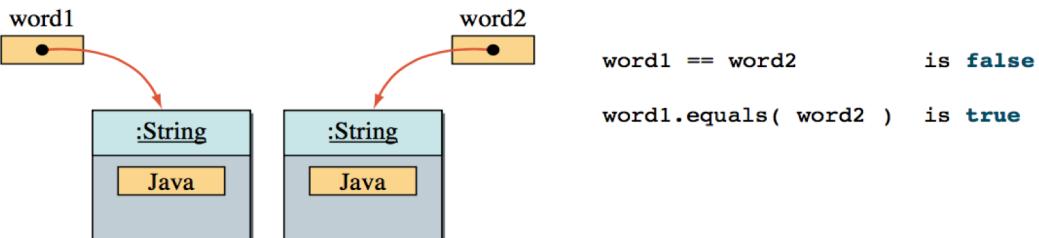
and

```
if ( word1.equals(word2) ) ...
```

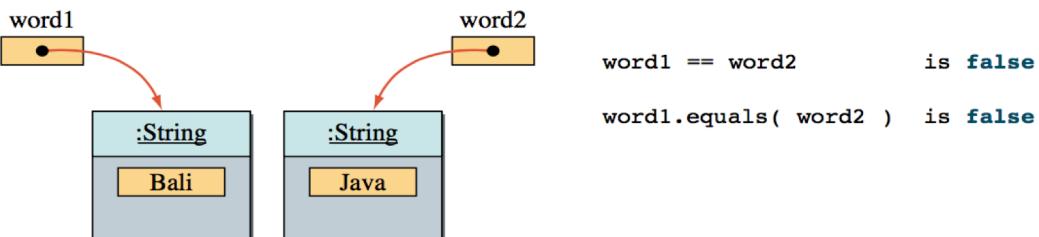
Case A: Referring to the same object.



Case B: Referring to different objects having identical string values.



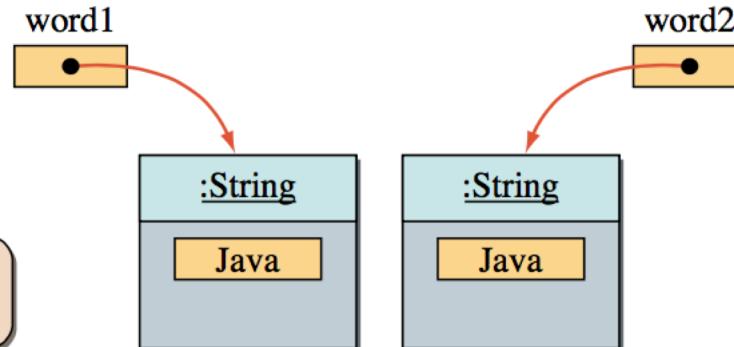
Case C: Referring to different objects having different string values.



Différence entre l'utilisation et la non-utilisation du new opérateur pour les chaînes de caractères

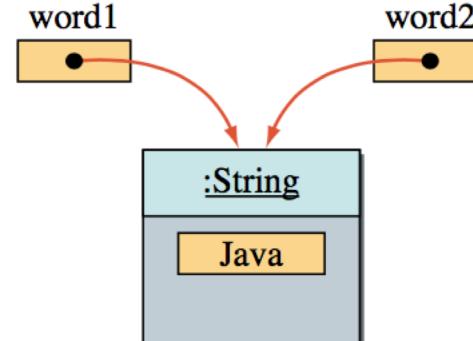
```
String word1, word2;  
  
word1 = new String("Java");  
  
word2 = new String("Java");
```

Whenever the **new** operator is used,
there will be a new object.



```
String word1, word2;  
  
word1 = "Java";  
  
word2 = "Java";
```

Literal string constant such as "Java" will
always refer to the one object.



- Un objet String est immuable, ce qui signifie qu'une fois qu'un objet String est créé, nous ne pouvons pas le modifier. En d'autres termes, nous pouvons lire les caractères individuels d'une chaîne, mais nous ne pouvons pas ajouter, supprimer ou modifier les caractères d'un objet String.
- les méthodes de la classe String, telles que replaceAll et substring, ne modifient pas la chaîne de caractères d'origine ; elles renvoient une nouvelle chaîne.
- Java adopte cette restriction d'immuabilité pour mettre en œuvre un schéma d'allocation de mémoire efficace pour la gestion des objets String. L'immuabilité est la raison pour laquelle nous pouvons traiter les données de type String comme un type de données primitif.

```
public static void main(String[] args)
{
    String s1 = "java";
    s1.concat(" rules");
    // Yes, s1 still refers to "java"
    System.out.println("s1 refers to " + s1);
}
```

Output : s1 refers to java

- Puisque les String sont immuables, la VM ne peut pas attribuer cette valeur à s1, elle crée donc un nouvel objet String, lui donne la valeur **java rules**.
- Les méthodes appliquées à un objet String afin de le modifier, créent un nouvel objet String.

- La création d'une nouvelle chaîne à partir de l'ancienne fonctionne dans la plupart des cas, mais il est parfois plus pratique de manipuler directement le contenu d'une chaîne.
- Lorsque nous devons composer une longue chaîne à partir d'un certain nombre de mots, par exemple, il est beaucoup plus pratique de pouvoir manipuler directement le contenu d'une chaîne que de créer une nouvelle copie de celle-ci.
- La manipulation des chaînes de caractères désigne ici des opérations telles que le remplacement d'un caractère, l'ajout d'une chaîne à une autre, la suppression d'une partie d'une chaîne, etc.
- Si nous devons manipuler le contenu d'une chaîne directement, nous devons utiliser la classe StringBuffer ou StringBuilder.

- Remarquez qu'aucune nouvelle chaîne n'est créée, la chaîne originale Java est modifiée. De plus, nous devons utiliser la méthode new pour créer un objet StringBuffer.

```
StringBuffer word = new StringBuffer( "Java" );
word.setCharAt(0, 'D');
word.setCharAt(1, 'i');
```

- Deux ou plusieurs méthodes peuvent porter le même nom dans une même classe si elles acceptent des arguments différents. Cette fonctionnalité est connue sous le nom de **surcharge de méthodes**.
- La surcharge des méthodes est obtenue soit en
 1. changeant le nombre d'arguments.
 2. changeant le type de données des arguments.
- Il ne s'agit pas d'une surcharge de méthodes si l'on change uniquement le type de retour des méthodes. Il doit y avoir des différences dans le nombre/type de paramètres.

Surcharge

- La surcharge se réfère à l'utilisation du même nom pour plus d'une méthode dans la même classe.
- Si vous essayez ce premier exemple dans Eclipse, vous verrez qu'il y a des erreurs indiquant que la méthode `read()` est dupliquée. Le problème est que vous avez deux méthodes avec le même nom et aucune n'a de paramètres.

```
public class Book {  
    String title;  
    String author;  
    boolean isRead;  
    int numberOfReadings;  
  
    public void read(){  
        isRead = true;  
    }  
  
    public void read(){  
        numberOfReadings++;  
    }  
}
```

Pour Java, il s'agit de deux méthodes complètement différentes, de sorte que vous n'aurez pas d'erreurs de duplication.

```
public class Book {  
    String title;  
    String author;  
    boolean isRead;  
    int numberOfReadings;  
  
    public void read(){  
        isRead = true;  
        numberOfReadings++;  
    }  
  
    public void read(int i){  
        isRead = true;  
        numberOfReadings += i;  
    }  
}
```

- La surcharge permet à différentes méthodes d'avoir le même nom, mais des signatures différentes où la signature peut différer par le nombre de paramètres d'entrée ou le type de paramètres d'entrée ou les deux.
- Plusieurs méthodes peuvent partager le même nom pour autant que l'une des règles suivantes soit respectée :
 1. Elles ont un nombre différent de paramètres.
 2. Les paramètres sont de types différents lorsque le nombre de paramètres est le même.
- Plus formellement, on dit que deux méthodes peuvent être surchargées si elles n'ont pas la même signature. **La signature de la méthode fait référence au nom de la méthode ainsi qu'au nombre et aux types de ses paramètres.**

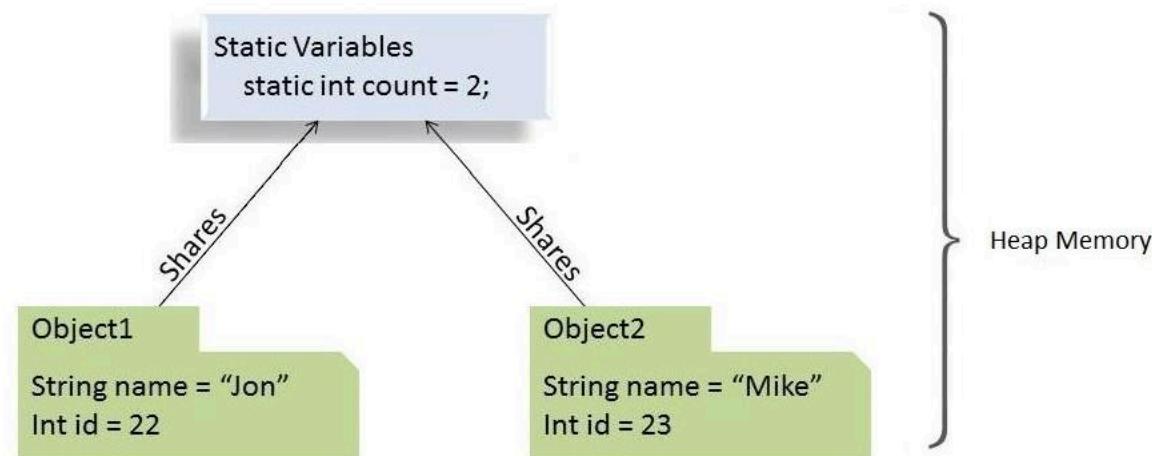
Overloading Constructors

- La définition de plusieurs constructeurs pour une classe offre au programmeur client une certaine souplesse dans la création des instances. Le programmeur client peut choisir l'un des différents constructeurs qui convient à ses besoins.

```
class Bicycle {  
    // Data Members  
    private String id;  
    private String ownerName;  
    // Constructors  
    public Bicycle( ) {  
        id = "XXXX-XXXX";  
        ownerName = "Unassigned";  
    }  
  
    public Bicycle(String tagNo, String name) {  
        id = tagNo;  
        ownerName = name;  
    }  
    //the rest of the class  
    ...  
}
```

Static

- le mot clé **static** signifie que le membre particulier appartient à un type lui-même, plutôt qu'à une instance de ce type.
- Cela signifie que nous ne créerons qu'une seule instance de ce membre statique qui sera partagée par toutes les instances de la classe.



Static

- Lorsque nous déclarons **un champ statique**, une seule et unique copie de ce champ est créée et partagée entre toutes les instances de cette classe.
- Peu importe le nombre de fois que nous initialisons une classe. Il n'y aura toujours qu'une seule copie du champ statique lui appartenant. La valeur de ce champ statique sera partagée entre tous les objets de la même classe ou d'une classe différente.
- Du point de vue de la mémoire, les variables statiques sont stockées dans la mémoire du tas.

Static

Pourquoi?

- Lorsque la valeur de la variable est indépendante des objets
- Lorsque la valeur est censée être partagée par tous les objets

Points clés à retenir

- Puisque les variables statiques appartiennent à une classe, nous pouvons y accéder directement en utilisant le nom de la classe. Nous n'avons donc pas besoin de référence à un objet.
- Nous ne pouvons déclarer des variables statiques qu'au niveau de la classe.
- Nous pouvons accéder aux champs statiques sans initialisation de l'objet.
- Enfin, nous pouvons accéder aux champs statiques en utilisant une référence d'objet (comme `ford.numberOfCars++`). Mais nous devrions éviter cela car il devient difficile de déterminer s'il s'agit d'une variable d'instance ou d'une variable de classe. Au lieu de cela, nous devrions toujours faire référence aux variables statiques en utilisant le nom de la classe (`Car.nombreDeVoitures++`).

Static

- Tout comme les champs statiques, les méthodes statiques appartiennent également à une classe au lieu de l'objet. On peut donc les appeler sans créer l'objet de la classe dans laquelle elles résident.
- Pour accéder/manipuler des variables statiques et d'autres méthodes statiques qui ne dépendent pas des objets.
- Les méthodes statiques sont largement utilisées dans les classes utilitaires et les classes d'aide (Maths, Collection..)

Static

- Les méthodes statiques ne peuvent pas utiliser les mots-clés this ou super.
- Les combinaisons suivantes de méthodes et de variables d'instance et de classe sont valables :
 - ✓ Les méthodes d'instance peuvent accéder directement aux méthodes d'instance et aux variables d'instance.
 - ✓ Les méthodes d'instance peuvent également accéder directement aux variables statiques et aux méthodes statiques.
 - ✓ Les méthodes statiques peuvent accéder à toutes les variables statiques et aux autres méthodes statiques.

Static

- Java permet de créer une classe dans une classe. Il s'agit d'un moyen efficace de regrouper des éléments utilisés à un seul endroit =>code plus organisé et plus lisible.
- L'architecture des classes imbriquées se divise en deux :
 - ✓ Les classes imbriquées que nous déclarons statiques sont appelées classes imbriquées statiques.
 - ✓ Les classes imbriquées qui ne sont pas statiques sont appelées classes internes.
- La principale différence entre ces deux types de classes est que les classes internes ont accès à tous les membres de la classe englobante (y compris les membres privés), alors que les classes statiques imbriquées n'ont accès qu'aux membres statiques de la classe externe.

Inheritance

- Parfois, on a besoin de la même fonctionnalité à plusieurs endroits de l'application.
- Une façon consiste à copier le même code à tous les endroits où on a besoin de la même fonctionnalité.
- Si on suit cette logique, on doit effectuer des modifications à tous les endroits lorsque la fonctionnalité change.
- **Maintenabilité**

Inheritance

- L'héritage est la caractéristique de la programmation orientée objet qui permet dans ces circonstances d'éviter de copier le même code à plusieurs endroits, facilitant ainsi **la réutilisation** du code.
- L'héritage permet également de personnaliser le code sans modifier le code existant.

Inheritance

- L'héritage est l'une des pierres angulaires des langages de programmation orientés objet.
- Elle permet de créer une nouvelle classe en réutilisant le code d'une classe existante. La nouvelle classe est appelée **sous-classe (classe fille)** et la classe existante est appelée **superclasse (classe mère)**.
- Une superclasse contient le code qui est réutilisé et personnalisé par la sous-classe. On dit que la sous-classe **hérite** de la **superclasse**.
- Une superclasse est également connue sous le nom de classe de base ou de classe mère, tandis qu'une sous-classe est également connue sous le nom de classe dérivée ou de classe enfant.

Inheritance

- “is-a” relationship.
- Avant d'hériter la classe Q de la classe P, on pose une question simple : "**Un objet de la classe Q est-il également un objet de la classe P ?**"
- "Un objet de la classe Q se comporte-t-il comme un objet de la classe P".
- Si la réponse est oui, la classe Q peut hériter de la classe P
- Par exemple, un manager est un type spécifique d'employé. Un employé est un type spécifique d'objet. Lorsque vous montez dans la hiérarchie de l'héritage, vous passez d'un type spécifique à un type plus général.

Inheritance

Comment l'héritage affecte-t-il le code client ?

- le code client est tout code qui utilise les classes dans une hiérarchie de classes.
- **L'héritage garantit que tout comportement présent dans une classe sera également présent dans sa sous-classe.** Une méthode dans une classe représente un comportement des objets de cette classe. Cela signifie que tout comportement qu'un code client attend d'une classe, sera également présent dans la sous-classe de cette classe.
- Cela conduit à la conclusion que si le code client fonctionne avec une classe, il fonctionnera également avec la sous-classe de la classe, car une sous-classe garantit **au moins** les mêmes comportements que sa super-classe.

Inheritance

The **extend** keyword

```
[modifiers] class <subclass-name> extends <superclass-name> {  
    // Code for the subclass goes here  
}
```

Inheritance

Employee est une classe simple avec une variable d'instance privée (name) et deux méthodes publiques, setName() et getName().

La variable d'instance est utilisée pour stocker le nom d'un employé et les deux méthodes sont utilisées pour obtenir et définir la variable d'instance name.

```
public class Employee {  
    private String name = "Unknown";  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

```
Employee emp = new Employee();  
emp.setName("John Jacobs");  
String empName = emp.getName();  
System.out.println("Employee Name: " + empName);
```

Inheritance

Manager class

Même la classe Manager ne contient aucun code, elle fonctionne de la même manière que la classe Employee, car elle hérite de la classe Employee. On crée un objet Manager en utilisant le constructeur de la classe Manager :

```
Manager mgr = new Manager();
```

Après la création de l'objet Manager, le code ressemble à celui utilisé pour traiter un objet Employee.

```
public class Manager extends Employee {  
    // No code is needed for now  
}  
  
public class SimplestInheritanceTest {  
    public static void main(String[] args) {  
        // Create an object of the Manager class  
        Manager mgr = new Manager();  
  
        // Set the manager name  
        mgr.setName("Leslie Zanders");  
  
        // Get the manager name  
        String mgrName = mgr.getName();  
  
        // Print the manager name  
        System.out.println("Manager Name: " + mgrName);  
    }  
}
```

Inheritance

Notez que la classe Manager **ne déclare pas** les méthodes setName() et getName().

Elle ne déclare pas non plus la variable d'instance name.

Lorsqu'une classe hérite d'une autre classe, elle hérite des membres de sa superclasse (variables d'instance, méthodes, etc.).

```
public class SimplestInheritanceTest {  
    public static void main(String[] args) {  
        // Create an object of the Manager class  
        Manager mgr = new Manager();  
  
        // Set the manager name  
        mgr.setName("Leslie Zanders");  
  
        // Get the manager name  
        String mgrName = mgr.getName();  
  
        // Print the manager name  
        System.out.println("Manager Name: " + mgrName);  
    }  
}
```

Inheritance

- La classe Object est la superclasse par défaut
- C'est la raison pour laquelle les objets de ces classes ont pu utiliser les méthodes de la classe Object.
- Exemple:

```
Employee emp = new Employee();
```

```
String str = emp.toString();
```

```
// #1 - "extends Object" is implicitly added for class P
public class P {
    // Code for class P goes here
}

// #2 - "extends Object" is explicitly added for class P
public class P extends Object {
    // Code for class P goes here
}
```

Inheritance

Inheritance and Constructors

- Contrairement aux autres membres d'une superclasse, **les constructeurs d'une superclasse ne sont pas hérités par ses sous-classes**. Cela signifie que vous devez définir un constructeur pour une classe ou utiliser le constructeur par défaut ajouté par le compilateur.

```
class Person {  
    public void sayHello( ) {  
        System.out.println("Well, hello.");  
    }  
}
```



```
class Person {  
    public Person( ) {  
        super(); ← This statement calls the  
        //superclass's constructor.  
    }  
    public void sayHello( ) {  
        System.out.println("Well, hello.");  
    }  
}
```

Automatically added to the class by the compiler →

Inheritance

super();

- Appelle le constructeur de la superclasse.
- Chaque classe a une superclasse.
- Si la déclaration de la classe ne désigne pas explicitement la superclasse avec la clause extends, alors la superclasse de la classe est la classe Object.

```
class Person {  
    public Person() {  
        super(); // This statement calls the  
        // superclass's constructor.  
    }  
  
    public void sayHello() {  
        System.out.println("Well, hello.");  
    }  
}
```

Automatically added to the class by the compiler → |

|

→ This statement calls the superclass's constructor.

Inheritance

Puisque la classe a un constructeur, aucun constructeur par défaut n'est ajouté à la classe. Cela signifie qu'une instruction telle que

```
Vehicle myCar = new Vehicle();
```

provoque une erreur de compilation car la classe ne possède pas de constructeur correspondant.

C'est en fait ce que nous voulons car nous ne voulons pas créer une instance de Vehicle sans numéro d'identification du véhicule.

```
class Vehicle {  
    private String vin;  
  
    public Vehicle(String vehicleIdNumber) {  
        vin = vehicleIdNumber;  
    }  
  
    public String getVIN() {  
        return vin;  
    }  
}
```

Inheritance

- Considérons maintenant une définition de sous-classe pour les camions (Truck).
- Un objet Truck possède une variable d'instance supplémentaire appelée cargoWeightLimit, qui fait référence au poids maximal de la cargaison que le camion peut transporter. Nous supposons que le poids maximal de la cargaison du camion peut varier (par exemple, en fonction du montant des frais payés par le propriétaire).

```
class Truck extends Vehicle {  
    private int cargoWeightLimit;  
    public void setWeightLimit(int newLimit) {  
        cargoWeightLimit = newLimit;  
    }  
    public int getWeightLimit() {  
        return cargoWeightLimit;  
    }  
}
```

Inheritance

```
public void Truck() {
    super();
}
```

Si on compile cette définition, nous obtiendrons une erreur de compilation. Comme aucun constructeur n'est défini pour la classe, le compilateur ajoute un constructeur par défaut. Ce constructeur appelle le constructeur de la superclasse sans argument, mais il n'y a pas de constructeur correspondant dans la superclasse. Ainsi, l'erreur de compilation résulte. Voici une définition correcte :

You need to make this call.
Otherwise, the compiler will add `super()`, which will result in an error because there is no matching constructor in `Vehicle`.

```
class Truck extends Vehicle {
    private int cargoWeightLimit;
    public Truck(int weightLimit, String vin) {
        → super(vin);
        cargoWeightLimit = weightLimit;
    }
    public void setWeightLimit(int newLimit) {
        cargoWeightLimit = newLimit;
    }
    public int getWeightLimit() {
        return cargoWeightLimit;
    }
}
```

```
class Vehicle {
    private String vin;
    public Vehicle(String vehicleIdNumber) {
        vin = vehicleIdNumber;
    }
}
```

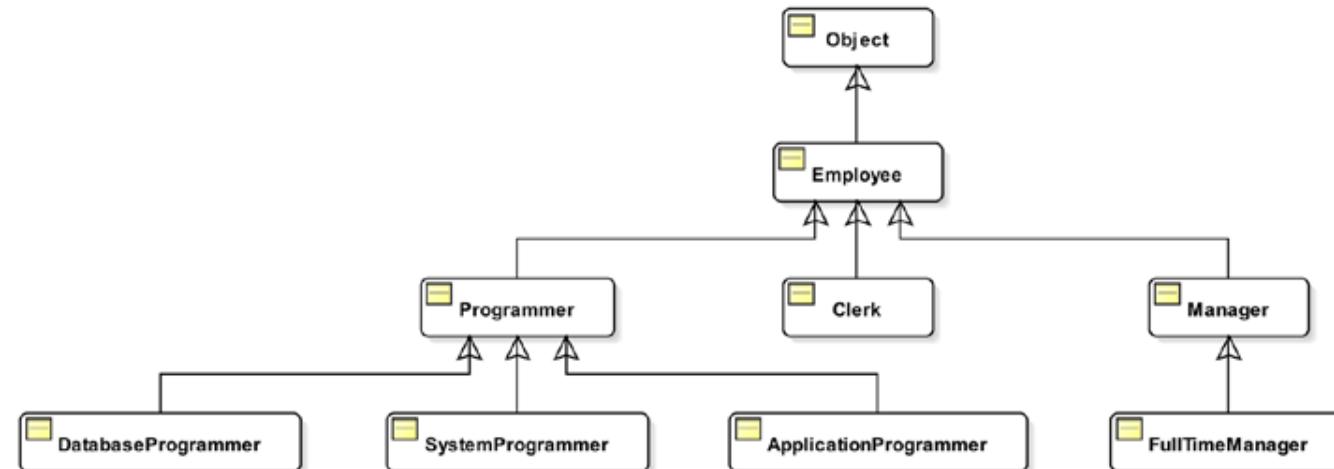
Inheritance

Héritage et relation hiérarchique

- l'héritage ne doit être utilisé que si une relation "is-a" existe entre la sous-classe et la super-classe.
- Une sous-classe peut avoir ses propres sous-classes, qui à leur tour peuvent avoir leurs propres sous-classes, et ainsi de suite.
- Toutes les classes d'une chaîne d'héritage forment une structure arborescente, appelée hiérarchie d'héritage ou hiérarchie de classes.
- Toutes les classes situées au-dessus d'une classe dans la hiérarchie d'héritage sont appelées **ancêtres** de cette classe. Toutes les classes situées au-dessous d'une classe dans la hiérarchie d'héritage sont appelées les **descendants** de cette classe.

Inheritance

Java autorise l'héritage unique pour une classe. C'est-à-dire qu'une classe ne peut avoir qu'une seule superclasse (ou parent). Toutefois, une classe peut être la superclasse de plusieurs classes. Toutes les classes de Java ont une superclasse, sauf la classe Object. La classe Object se trouve au sommet des hiérarchies d'héritage.



Inheritance

- Il existe quatre modificateurs d'accès : private, public, protected, and package-level.
- L'absence de modificateur d'accès private, public et protected est considérée comme un accès par défaut (package).
- Le modificateur de niveau d'accès d'un membre de classe détermine deux choses :
 - 1.Qui peut accéder (ou utiliser) directement ce membre de classe.
 2. Si une sous-classe hérite ou non de ce membre de classe.
- Si un membre de classe est déclaré privé, il n'est accessible qu'à l'intérieur de la classe qui le déclare. Un membre de classe privé n'est pas hérité par les sous-classes de cette classe.
- Un membre de classe public est accessible de partout, à condition que la classe elle-même soit accessible. Une sous-classe hérite de tous les membres publics de sa super-classe.

Inheritance

Remarquez que les deux classes Super et Sub sont placées dans des packages distincts. Nous devons placer la classe Sub dans un package différent de celui de sa superclasse pour montrer l'effet du modificateur protected. *Si la superclasse et la sous-classe se trouvent dans le même package, le modificateur protected n'a aucun effet (il se comporte comme le modificateur public).*

```
package one;
class Super {
    public int public_Super_Field;
    protected int protected_Super_Field;
    private int private_Super_Field;

    public Super() {
        public_Super_Field = 10;
        protected_Super_Field = 20;
        private_Super_Field = 30;
    }
    ...
}
```

super is a reserved word, so
don't use it.

```
package two;
import one.*;
class Sub extends Super {
    public int public_Sub_Field;
    protected int protected_Sub_Field;
    private int private_Sub_Field;

    public Sub() {
        public_Sub_Field = 100;
        protected_Sub_Field = 200;
        private_Sub_Field = 300;
    }
    ...
}
```

Inheritance

Les membres publics d'une classe, qu'ils soient hérités ou non, sont accessibles depuis n'importe quel objet ou classe.

```
import one.*;
import two.*;

class Client {
    public void test() {
        Super mySuper = new Super();
        Sub   mySub = new Sub();

        int i = mySuper.public_Super_Field;
        ✓ VALID → | int j = mySub.public_Super_Field; //inherited
                    |                         //by mySub
        int k = mySub.public_Sub_Field;
    }
}
```

Inheritance

Les membres privés d'une classe, en revanche, ne sont jamais accessibles depuis un objet ou une classe extérieure. Les déclarations suivantes, si elles sont placées dans la classe Client, sont donc toutes invalides :

X NOT VALID →

```
int l = mySuper.private_Super_Field;  
int m = mySub.private_Super_Field;  
int n = mySub.private_Sub_Field;
```

Inheritance

- Outre les modificateurs **private**, **protected** et **public**, Java prend en charge un quatrième modificateur de visibilité, appelé **package visibility**.
- Si aucun modificateur explicite (public, private et protected) n'est inclus dans la déclaration, le composant est visible par package, ce qui signifie que le composant est accessible à partir de toute classe appartenant au même package que la classe du composant.
- Les composants **protected** sont accessibles à toutes les classes du même package, mais ils sont également accessibles à toutes les sous-classes en dehors du package.
- Les composants visibles par le package sont inaccessibles à toutes les classes en dehors du paquetage, la visibilité protégée est donc moins restrictive que la visibilité par le paquetage.

Inheritance

Access Modifier	Within Class	Within Package	Same Package by subclasses	Outside Package by subclasses	Global
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

Inheritance

- Ce bout de code se compile sans aucune erreur.
- Lorsque le compilateur rencontre les appels `emp.setName("Richard Castillo")` et `emp.getName()`, il vérifie **le type déclaré** de la variable `emp`.
- Il constate que le type déclaré de la variable `emp` est `Employee`. Il s'assure que la classe `Employee` possède des méthodes `setName()` et `getName()` conformes à l'appel effectué.
- Il constate que la classe `Employee` possède une méthode `setName()` qui prend une chaîne de caractères comme paramètre. Il constate que la classe `Employee` possède bien une méthode `getName()` qui ne prend aucun paramètre et renvoie une chaîne de caractères. Après avoir vérifié ces deux faits, le compilateur accepte les appels des méthodes `emp.setName()` et `emp.getName()`.

```
Employee emp;  
emp = new Employee();  
emp.setName("Richard Castillo");  
String name = emp.getName();
```

Inheritance

```
Employee emp;  
emp = new Manager(); // A Manager object assigned to an Employee variable  
emp.setName("Richard Castillo");  
String name = emp.getName();
```

- Le compilateur compilera également ce bout de code, même si on a attribué à la variable emp un objet de la classe Manager.
- Puisque la classe Manager hérite de la classe Employee, un objet de la classe Manager "est" un objet de la classe Employee.
- En clair, on dit qu'un manager est toujours un employé. Une telle affectation (de la sous-classe à la superclasse) est appelée **upcasting** et elle est toujours autorisée en Java. On l'appelle également une conversion d'élargissement (widening conversion) car un objet de la classe Manager (type plus spécifique) est affecté à une variable de référence du type Employee (type plus générique).

Inheritance

Si le type déclaré (compile-time) de l'opérande de droite est une sous-classe du type déclaré de l'opérande de gauche, il s'agit d'un cas d'upcasting, et l'affectation est sûre et autorisée.

L'upcasting est une traduction technique directe du fait qu'un objet d'une sous-classe "est aussi" un objet de la super-classe.

```
// An employee is always an object  
obj = emp;  
  
// A manager is always an employee  
emp = mgr;  
  
// A part-time manager is always a manager  
mgr = ptm;  
  
// A part-time manager is always an employee  
emp = ptm;  
  
// A part-time manager is always an object  
obj = ptm;
```

Inheritance

L'affectation d'une référence de superclasse à une variable de sous-classe est appelée downcasting (ou conversion restrictive). Le downcasting est le contraire du upcasting. Dans le cas de l'upcasting, l'affectation se déplace vers le haut de la hiérarchie des classes, alors que dans le cas du downcasting, l'affectation se déplace vers le bas de la hiérarchie des classes.

```
Employee emp;  
Manager mgr = new Manager();  
emp = mgr; // OK. Upcasting  
mgr = emp; // A compile-time error. Downcasting
```

Inheritance

- L'affectation `emp = mgr` est autorisée en raison de l'upcasting.
- Cependant, l'affectation `mgr = emp` n'est pas autorisée car il s'agit d'un cas de downcasting où une variable de superclasse (`Employee`) est affectée à une variable de sous-classe (`Manager`).
- Le compilateur a raison de supposer que tous les managers sont des employés (upcasting). Cependant, tous les employés ne sont pas des managers (downcasting).

```
Employee emp;  
Manager mgr = new Manager();  
emp = mgr; // OK. Upcasting  
mgr = emp; // A compile-time error. Downcasting
```

Inheritance

- En supposant qu'on a une sous-classe de la classe Manager, qui s'appelle PartTimeManager.
- La dernière affectation, qui utilise le downcasting, réussit au moment de la compilation car le type déclaré de la variable ptm et le type typecast sont les mêmes. Le type d'exécution de emp est Manager, car l'instruction emp = mgr lui attribue la référence d'un objet Manager.
- Lorsque le runtime tente d'exécuter la partie "(PartTimeManager) emp" du downcasting, il constate que le type d'exécution de emp, qui est Manager, n'est pas compatible avec le type de typecast, qui est PartTimeManager. C'est la raison pour laquelle le runtime lèvera une ClassCastException.

```
Employee emp;
Manager mgr = new Manager();
PartTimeManager ptm = new PartTimeManager();

emp = mgr; // Upcasting. OK
ptm = (PartTimeManager) emp; // Downcasting. OK at compile-time. A runtime error.
```

Inheritance

- La liaison (Binding) est le processus d'identification du code de la méthode accédée (maMéthode() dans ce cas) ou du champ (xyz dans ce cas), qui sera utilisé lorsque le code sera exécuté.
- En d'autres termes, la liaison est un processus qui consiste à prendre une décision sur le code ou le champ de la méthode qui sera utilisé lors de l'exécution du code. La liaison peut se faire à deux stades : au moment de la compilation et au moment de l'exécution.

```
MyClass myobject = get an object reference;  
myObject.myMethod(); // Which myMethod() to call?  
int a = myObject.xyz; // Which xyz to access?
```

Inheritance

- Lorsque la liaison se produit au moment de la compilation, elle est appelée liaison précoce. La liaison précoce est également connue sous le nom de liaison statique ou liaison au moment de la compilation.
- Lorsque la liaison se produit au moment de l'exécution, on parle de liaison tardive. La liaison tardive est également appelée liaison dynamique ou liaison au moment de l'exécution.

```
MyClass myobject = get an object reference;  
myObject.myMethod(); // Which myMethod() to call?  
int a = myObject.xyz; // Which xyz to access?
```

Inheritance

- Dans la liaison précoce, la décision concernant le code et le champ de la méthode à laquelle on accédera est prise par le compilateur au moment de la compilation. Pour un appel de méthode, le compilateur décide quelle méthode de quelle classe sera exécutée lorsque le code comportant l'appel de méthode est exécuté. Pour un accès à un champ, le compilateur décide quel champ de quelle classe sera accédé lorsque le code ayant l'accès au champ est exécuté. La liaison anticipée est utilisée pour les types de méthodes et de champs suivants d'une classe :
 - Tous les types de champs : statiques et non statiques
 - Méthodes statiques
 - Méthodes finales non statiques

Polymorphisme

- Le polymorphisme en Java est un concept par lequel nous pouvons effectuer une même action de différentes manières. Le polymorphisme est dérivé de 2 mots grecs : poly et morphs. Le mot "poly" signifie beaucoup et "morphs" signifie formes. Le polymorphisme signifie donc de nombreuses formes.
- Il existe deux types de polymorphisme en Java : le polymorphisme à la compilation et le polymorphisme à l'exécution.
- Nous pouvons réaliser le polymorphisme en Java par la surcharge de méthodes et le redéfinition de méthodes.

Polymorphisme

Polymorphisme statique

- Le polymorphisme statique est également connu sous le nom de liaison au moment de la compilation ou liaison précoce.
- La liaison statique se produit au moment de la compilation. La surcharge de méthode est un exemple de liaison statique où la liaison de l'appel de la méthode à sa définition se produit au moment de la compilation.

Polymorphisme

Dynamic binding

- La liaison de toutes les méthodes *non statiques* et *non finales* suit les règles de la **liaison tardive**.
- En d'autres termes, si votre code accède à une méthode non statique, qui n'est pas déclarée finale, la décision quant à la version de la méthode à appeler est prise au moment de l'exécution. La version de la méthode qui sera appelée dépend du type d'exécution de l'objet sur lequel l'appel de méthode est effectué, et non de son type de compilation.

Polymorphisme

```
Employee emp = new Manager();  
emp.setName("John Jacobs");
```

- Le compilateur effectue une seule vérification pour l'appel de la méthode emp.setName() dans ce code.
- Il s'assure que le type déclaré de la variable emp, qui est Employee, possède une méthode appelée setName(String s). Le compilateur détecte que la méthode setName(String s) de la classe Employee est une méthode d'instance, qui n'est pas finale.
- Pour un appel à une méthode d'instance, le compilateur n'effectue pas de liaison. Il laisse ce travail au runtime. L'appel de la méthode emp.setName("John Jacobs") est un exemple de liaison tardive.

Polymorphisme

```
Employee emp = new Manager();
emp.setName("John Jacobs");
```

- Au moment de l'exécution, la JVM décide quelle méthode setName(String s) doit être appelée. La JVM obtient le type d'exécution de la variable emp. Le type d'exécution de la variable emp est Manager lorsque l'instruction emp.setName("John Jacobs") est examinée dans ce code.
- La JVM remonte la hiérarchie des classes en partant du type d'exécution (c'est-à-dire Manager) de la variable emp, à la recherche de la définition d'une méthode setName(String s). Elle commence par examiner la classe Manager et constate que celle-ci ne déclare pas de méthode setName(String s). La JVM remonte maintenant d'un niveau dans la hiérarchie des classes, à savoir la classe Employee. Elle constate que la classe Employee déclare une méthode setName(String s).
- Dès que la JVM trouve une correspondance, elle lie l'appel à cette méthode et arrête la recherche.
- Rappelons que la classe Object se trouve toujours au sommet de toutes les hiérarchies de classes en Java. La JVM poursuit sa recherche d'une définition de méthode jusqu'à la classe Object. Si elle ne trouve pas de correspondance dans la classe Object, elle lève une exception d'exécution.

Polymorphisme

Redéfinition des méthodes/ overriding

La redéfinition d'une méthode d'instance dans une classe, qui est héritée de la superclasse

La classe B est une sous-classe de la classe A. La classe B hérite de la méthode print() de sa super-classe et la redéfinit. On dit que la méthode print() de la classe B overrides/redéfinit la méthode print() de la classe A.

```
public class A {  
    public void print() {  
        System.out.println("A");  
    }  
}  
  
public class B extends A {  
    @Override  
    public void print() {  
        System.out.println("B");  
    }  
}
```

Polymorphisme

La classe C ne déclare aucune méthode. De quelle méthode la classe C hérite-t-elle : A.print() ou B.print(), ou les deux ?

Elle hérite de la méthode print() de la classe B. Une classe hérite toujours de ce qui est disponible dans sa superclasse immédiate (déclaré dans la superclasse ou hérité par sa superclasse). Si une classe D hérite de la classe C, elle héritera de la méthode print() de la classe B par l'intermédiaire de la classe C.

```
public class C extends B {  
    // C inherits B.print()  
}
```

Polymorphisme

```
public class A {  
    public void print() {  
        System.out.println("A");  
    }  
}  
  
public class B extends A {  
    @Override  
    public void print() {  
        System.out.println("B");  
    }  
}  
  
public class C extends B {  
    // C inherits B.print()  
}  
  
public class D extends C {  
    // D inherits B.print() through C  
}
```

```
public class E extends D {  
    @Override  
    public void print() {  
        System.out.println("E");  
    }  
}  
  
public class F extends E {  
    // F inherits E.print() through E  
}
```

```
A a=new A();  
a.print();  
a= new B();  
a.print();  
a=new C();  
a.print();  
a=new D();  
a.print();  
a= new E();  
a.print();
```

Polymorphisme

```
public class A {  
    public void print() {  
        System.out.println("A");  
    }  
}  
  
public class B extends A {  
    @Override  
    public void print() {  
        System.out.println("B");  
    }  
}  
  
public class C extends B {  
    // C inherits B.print()  
}  
  
public class D extends C {  
    // D inherits B.print() through C  
}
```

```
public class E extends D {  
    @Override  
    public void print() {  
        System.out.println("E");  
    }  
}  
  
public class F extends E {  
    // F inherits E.print() through E  
}
```

```
A a=new A();  
a.print(); //A  
a= new B();  
a.print(); //B  
a=new C();  
a.print(); //B  
a=new D();  
a.print(); //B  
a= new E();  
a.print(); //E
```

Polymorphisme

La méthode print() de la classe T redéfinit la méthode print() de sa superclasse S ?

La réponse est non. La méthode print() de la classe T ne redéfinit pas la méthode print() de la classe S. C'est **une surcharge** de méthodes.

La classe T aura maintenant deux méthodes print() : une héritée de sa superclasse S, qui ne prend aucun argument, et une déclarée dans la classe, qui prend un argument String. Cependant, les deux méthodes de la classe T ont le même nom print. C'est la raison pour laquelle on parle de surcharge de méthodes, car le même nom de méthode est utilisé plus d'une fois dans la même classe.

```
public class S {  
    public void print() {  
        System.out.println("S");  
    }  
}
```

```
public class T extends S {  
    public void print(String msg) {  
        System.out.println(msg);  
    }  
}
```

Polymorphisme

- Règle n° 1 La méthode doit être une méthode d'instance. La redéfinition ne s'applique pas aux méthodes statiques.
- Règle n° 2 La méthode redéfinie doit avoir le même nom que la méthode à redéfinir.
- Règle n° 3 La méthode redéfinie doit avoir le même nombre de paramètres du même type et dans le même ordre que la méthode à redéfinir.
- Notez que le nom du paramètre n'a pas d'importance. Par exemple, void print(String str) et void print(String msg) sont considérés comme la même méthode. Les noms différents des paramètres, str et msg, n'en font pas des méthodes différentes.

Polymorphisme

- Le niveau d'accès de la méthode redéfinie doit être au moins le même ou plus souple que celui de la méthode à redéfinir.
- Les trois niveaux d'accès sont public, protected et package qui permet l'héritage. Rappelons que les membres privés ne sont pas hérités et ne peuvent donc pas être remplacés. L'ordre des niveaux d'accès, du plus souple au plus strict, est le suivant : public, protected et package-level.

Overridden Method Access Level	Allowed Overriding Method Access Level
public	public
protected	public, protected
package-level	public, protected, package-level

Polymorphisme

- Redéfinition des méthodes statiques
- Les méthodes statiques sont liées pendant la compilation en utilisant le type de variable de référence, et non l'objet. Si vous utilisez un IDE comme Netbeans et Eclipse, et si vous essayez d'accéder aux méthodes statiques en utilisant un objet, vous verrez des avertissements. Selon la convention de codage Java, les méthodes statiques doivent être accessibles par le nom de la classe plutôt que par un objet.
- En bref, une méthode statique peut être surchargée, mais ne peut pas être redéfinie en Java. Si vous déclarez une autre méthode statique avec la même signature dans une classe dérivée, la méthode statique de la superclasse sera cachée, et tout appel à cette méthode statique dans la sous-classe ira à la méthode statique déclarée dans cette classe elle-même. C'est ce qu'on appelle le "method hiding" en Java.

Polymorphisme

- Nous pouvons déclarer des méthodes statiques avec la même signature dans la sous-classe, mais ce n'est pas considéré comme une surcharge car il n'y aura pas de polymorphisme à l'exécution.
- Si une classe dérivée définit une méthode statique avec la même signature qu'une méthode statique de la classe de base, la méthode de la classe dérivée est cachée par la méthode de la classe de base.

Polymorphisme

Surcharge des méthodes

- Avoir plus d'une méthode avec le même nom dans une même classe est appelé surcharge de méthodes. Les méthodes surchargées doivent avoir un nombre différent de paramètres, des types différents de paramètres, ou les deux. Le type de retour, le niveau d'accès et la clause throws d'une méthode ne jouent aucun rôle pour en faire une méthode surchargée. La méthode m1() de la classe OME1 est un exemple de méthode surchargée.

```
public class OME1 {  
    public void m1(int a) {  
        // Code goes here  
    }  
  
    public void m1(int a, int b) {  
        // Code goes here  
    }  
  
    public int m1(String a) {  
        // Code goes here  
    }  
  
    public int m1(String a, int b) throws CheckedException1 {  
        // Code goes here  
    }  
}
```

Polymorphisme

- Peut-on surcharger les méthodes statiques ?
- **OUI**
- Nous pouvons avoir deux ou plusieurs méthodes statiques portant le même nom, mais dont les paramètres d'entrée sont différents.

Polymorphisme

	Surcharge	Redéfinition
1)	La surcharge de méthode est utilisée pour améliorer la lisibilité du programme.	La redéfinition de méthode est utilisée pour fournir l'implémentation spécifique de la méthode qui est déjà fournie par sa super classe.
2)	La surcharge de méthode est effectuée dans la classe elle même.	La redéfinition de méthode se produit dans deux classes ayant une relation d'héritage.
3)	En cas de surcharge de méthode, les paramètres doivent être différent.	En cas de redéfinition de méthode, les paramètres doivent être identique.
4)	La surcharge de méthode est l'exemple du polymorphisme au moment de la compilation.	La redéfinition de méthode est l'exemple du polymorphisme au moment de l'exécution.
5)	En Java, la surcharge de méthode ne peut pas être effectuée en modifiant uniquement le type de retour de la méthode. Le type de retour peut être identique ou différent dans la surcharge de méthode. Mais vous devez changer le paramètre.	Le type de retour doit être identique ou covariant lors de la redéfinition de méthode.

Polymorphisme

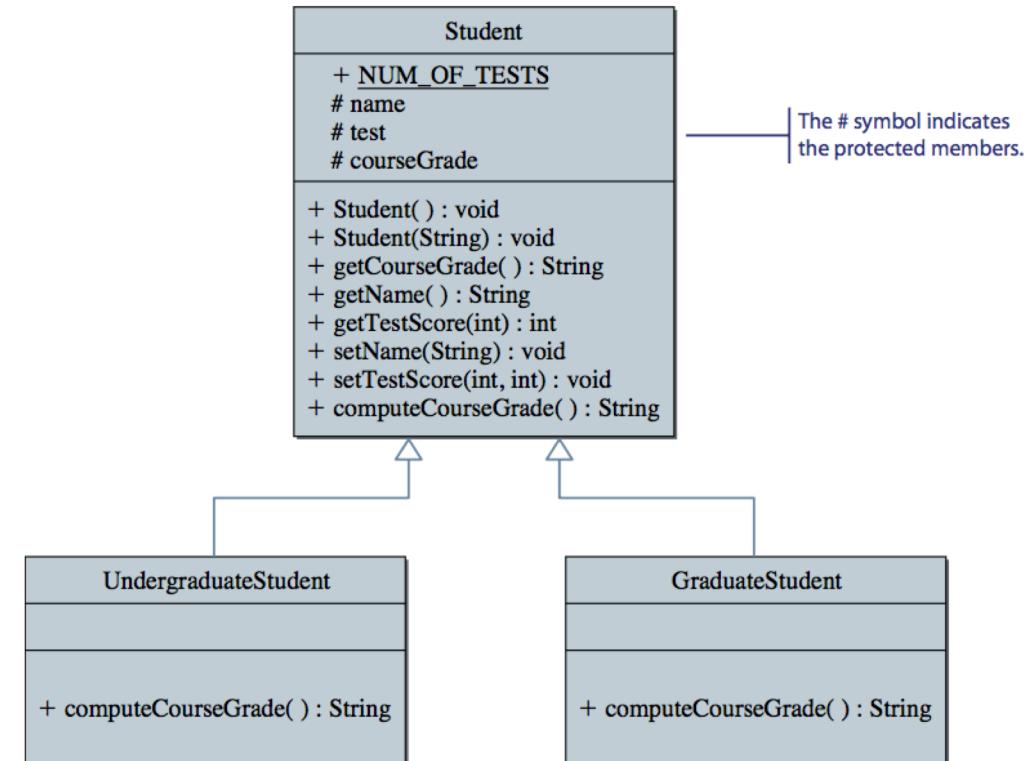
Utilisation efficace des classes grâce au polymorphisme

Voyons maintenant comment la classe Student et ses sous-classes peuvent être utilisées efficacement dans le programme de liste de classes.

Puisque les étudiants de premier et de second cycle sont inscrits dans une classe, devons-nous déclarer les deux tableaux ci-dessous pour maintenir la liste des étudiants ?

```
GraduateStudent gradRoster[20];
```

```
UndergraduateStudent undergradRoster[20];
```



Polymorphisme

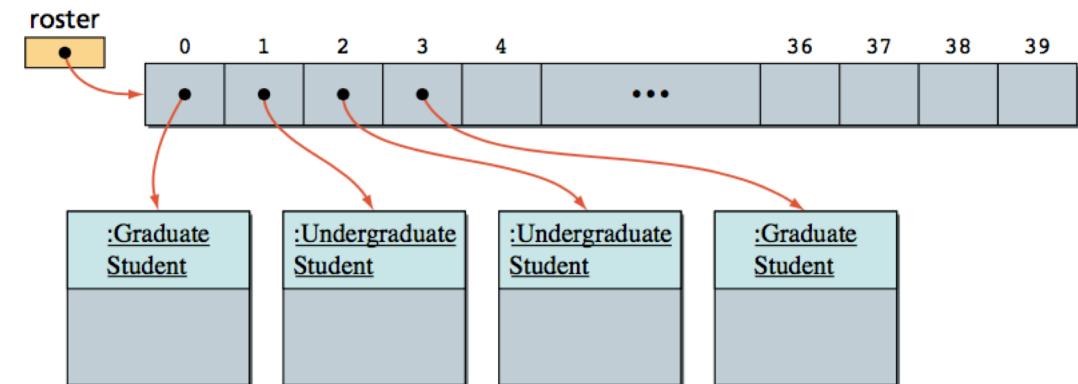
- Un tableau doit contenir des éléments du même type de données. Par exemple, nous ne pouvons pas stocker des nombres entiers et des nombres réels dans le même tableau.
- Pour respecter cette règle, il semble nécessaire de déclarer deux tableaux distincts, l'un pour les étudiants diplômés et l'autre pour les étudiants de premier cycle. Cette règle ne s'applique toutefois pas lorsque les éléments du tableau sont des objets. Il suffit alors de déclarer un seul tableau, par exemple,
- Student roster[40];

Polymorphisme

- Les éléments du tableau de listes peuvent être des instances de la classe Student ou de l'une de ses classes descendantes GraduateStudent ou UndergraduateStudent.
- Le polymorphisme permet à une seule variable de faire référence à des objets de différentes classes.

Student student;

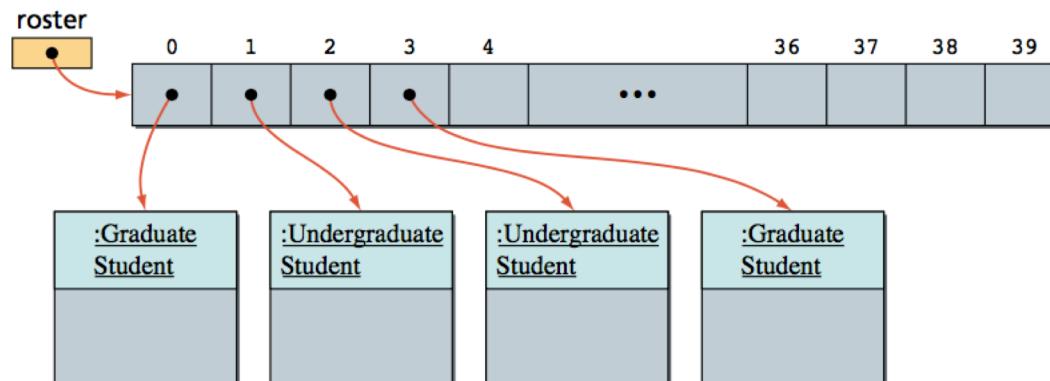
- ✓ student = new Student();
- ✓ student = new GraduateStudent();
- ✓ student = new UndergraduateStudent();



Polymorphisme

En d'autres termes, la variable unique student ne se limite pas à faire référence à un objet de la classe Student mais peut faire référence à tout objet des classes descendantes de Student.

```
roster[0] = new GraduateStudent();
roster[1] = new UndergraduateStudent();
roster[2] = new UndergraduateStudent();
roster[3] = new GraduateStudent();
...
...
```



Polymorphisme

Si roster[i] fait référence à un GraduateStudent, alors la méthode computeCourseGrade de la classe GraduateStudent est exécutée ; et s'il fait référence à un UndergraduateStudent, alors la méthode computeCourseGrade de UndergraduateStudent est exécutée. Nous appelons le message computeCourseGrade polymorphe parce que le message fait référence à des méthodes de différentes classes selon l'objet référencé par roster[i].

Le polymorphisme nous permet de maintenir la liste des classes avec un seul tableau au lieu de maintenir un tableau séparé pour chaque type d'étudiant, ce qui simplifie énormément le processus.

```
for (int i = 0; i < numberOfStudents; i++) {  
    roster[i].computeCourseGrade();  
}
```

Polymorphisme

- Le polymorphisme permet d'étendre et de modifier facilement un programme.
- Supposons, par exemple, que nous devions ajouter un troisième type d'élève, disons un élève auditeur, au programme de la liste des élèves.
- Si nous devons définir un tableau distinct pour chaque type d'élève, cette extension nous oblige à définir une nouvelle classe et un troisième tableau pour les élèves auditeurs. Mais avec le polymorphisme, il nous suffit de définir une nouvelle sous-classe de Student. Et tant que cette nouvelle sous-classe inclut la bonne méthode computeCourseGrade, la boucle for pour calculer la note de cours des étudiants reste la même.

Polymorphisme

- Sans le polymorphisme, nous devons non seulement ajouter le nouveau code, mais aussi réécrire le code existant pour tenir compte du changement.
- Avec le polymorphisme, en revanche, nous ne devons pas toucher au code existant. La modification du code existant est une activité fastidieuse et sujette aux erreurs. Une légère modification du code existant peut empêcher un programme de fonctionner correctement.

Polymorphisme

Un élément du tableau roster est une référence à une instance de la classe GraduateStudent ou UndergraduateStudent.

Si nous avons besoin de connaître la classe d'un objet référencé. Par exemple, nous pouvons vouloir connaître le nombre d'étudiants de premier cycle qui ont réussi le cours. Pour déterminer la classe d'un objet, nous utilisons l'opérateur **instanceof**.

```
Student x = new UndergraduateStudent( );
if ( x instanceof UndergraduateStudent ) {
    System.out.println("Mr. X is an undergraduate student");
} else {
    System.out.println("Mr. X is a graduate student");
}

int undergradCount = 0;
for (int i = 0; i < numberOfStudents; i++) {
    if ( roster[i] instanceof UndergraduateStudent ) {
        undergradCount++;
    }
}
```

- abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.
- In Java, abstraction is achieved using Abstract classes and interfaces.

Classes abstraites

- Une classe abstraite en Java est une classe déclarée avec le mot clé `abstract`.
- Elle peut avoir des méthodes abstraites et non-abstraites (méthodes avec corps).
- `Abstract` est un modificateur java applicable aux classes et aux méthodes en java mais pas aux Variables.

Classes abstraites

- Une classe qui contient le mot clé abstract dans sa déclaration est appelée classe abstraite.
- Les classes abstraites peuvent ou non contenir des méthodes abstraites, c'est-à-dire des méthodes sans corps (public void get() ;)
- Mais, si une classe a au moins une méthode abstraite, alors la classe doit être déclarée abstraite.
- Si une classe est déclarée abstraite, elle ne peut pas être instanciée.
- Pour utiliser une classe abstraite, vous devez en hériter d'une autre classe et lui fournir des implémentations des méthodes abstraites.
- Si vous héritez d'une classe abstraite, vous devez fournir des implémentations à toutes les méthodes abstraites qu'elle contient.
- Si la classe enfant est incapable de fournir l'implémentation de toutes les méthodes abstraites de la classe parent, nous devons déclarer cette classe enfant comme abstraite afin que la classe enfant de niveau suivant fournit l'implémentation de la méthode abstraite restante.

Classes abstraites

Au lieu des accolades, une méthode abstraite aura un point-virgule (;) à la fin.

```
public abstract class Employee {  
    private String name;  
    private String address;  
    private int number;  
  
    public abstract double computePay();  
    // Remainder of class definition  
}
```

Interface

- Une interface est une "**classe complètement abstraite**" qui sert à regrouper des méthodes apparentées ayant un corps vide
- Une interface en langage de programmation Java est définie comme un type abstrait utilisé pour spécifier le comportement d'une classe.
- Une interface en Java est le plan d'un comportement.
- Une interface Java contient des constantes statiques et des méthodes abstraites.

Interface

- L'interface en Java est un mécanisme permettant de réaliser l'abstraction.
- Il ne peut y avoir que des méthodes abstraites dans l'interface Java, pas le corps de la méthode.
- Elle est utilisée pour réaliser l'abstraction et l'héritage multiple en Java.
- Une interface peut avoir des méthodes et des variables, mais les méthodes déclarées dans une interface sont par défaut abstraites (seulement la signature de la méthode, pas le corps).
- Si une classe implémente une interface et ne fournit pas de corps de méthode pour toutes les fonctions spécifiées dans l'interface, alors la classe doit être déclarée abstraite.

Exceptions

- Les erreurs sont presque inévitables en programmation.
- Ces erreurs peuvent être appelées bogues, erreurs ou exceptions.
- Elles sont classées ici en trois catégories principales : les erreurs de syntaxe, les erreurs d'exécution et les erreurs logiques.

Les erreurs syntaxiques sont l'équivalent en programmation des fautes d'orthographe et de grammaire dans les langues naturelles. Les erreurs syntaxiques comprennent les exemples suivants :

- Noms de classe, de variable ou de méthode mal écrites
- Mots clés mal écrits
- Points-virgules manquants
- Type de retour manquant pour les méthodes
- Parenthèses et crochets mal placés ou mal assortis variables non déclarées ou non initialisées
- Format incorrect des boucles, méthodes ou autres structures.

```
public class errors {

    public static void main(String[] args) {
        age = 30;
        int retirementFund = 10000;
        int yearsInRetirement = 0;
        String name = "David Johnson",
        for (int i = age; <= 65; ++){
            recalculate(retirementFund,0.1);
        }
        int monthlyPension = retirementFund/yearsInRetirement/12
        System.out.println(name + " will have $" + monthlyPension
                           + " per month for retirement.");
    }

    public static recalculate(fundAmount, rate){
        fundAmount = fundAmount*(1+rate);
    }
}
```

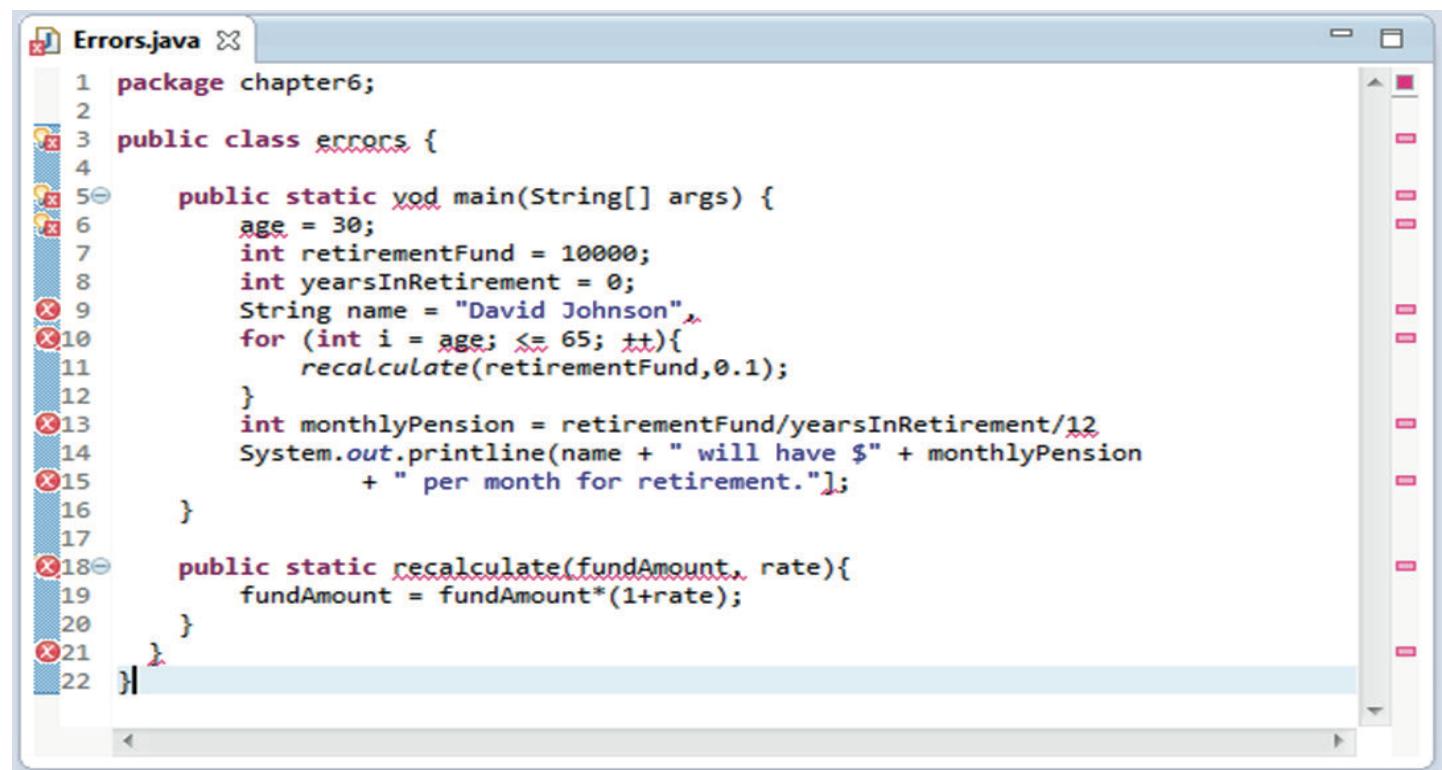
```
public class errors {

    public static void main(String[] args) {
        age = 30;
        int retirementFund = 10000;
        int yearsInRetirement = 0;
        String name = "David Johnson";
        for (int i = age; <= 65; ++){
            recalculate(retirementFund,0.1);
        }
        int monthlyPension = retirementFund/yearsInRetirement/12
        System.out.println(name + " will have $" + monthlyPension
                           + " per month for retirement.");
    }

    public static recalculate(fundAmount, rate){
        fundAmount = fundAmount*(1+rate);
    }
}
```

Age cannot be resolved to a variable

indique qu'une variable appelée age n'a pas encore été déclarée. Par conséquent, vous ne pouvez pas lui attribuer une valeur.



A screenshot of a Java code editor window titled "Errors.java". The code contains several syntax errors, indicated by red X icons next to the line numbers in the margin:

```
1 package chapter6;
2
3 public class errors {
4
5     public static void main(String[] args) {
6         age = 30;
7         int retirementFund = 100000;
8         int yearsInRetirement = 0;
9         String name = "David Johnson";
10        for (int i = age; i <= 65; i++) {
11            calculate(retirementFund, 0.1);
12        }
13        int monthlyPension = retirementFund / yearsInRetirement / 12;
14        System.out.println(name + " will have $" + monthlyPension
15                           + " per month for retirement.");
16    }
17
18    public static calculate(fundAmount, rate) {
19        fundAmount = fundAmount * (1 + rate);
20    }
21
22 }
```

Ligne 3 : Renommez le type en Errors (le nom de la classe doit correspondre au nom de fichier .java).

Ligne 5 : Changez en void (le type de retour de la méthode main est toujours void).

Ligne 9 : Remplacez la virgule par un point-virgule.

Ligne 10 : remplacez la boucle for par for (int i = age ; i <= 65 ; i++).

Ligne 18 : Définissez le type de retour de la méthode à void (la méthode recalculate ne renvoie aucune valeur).

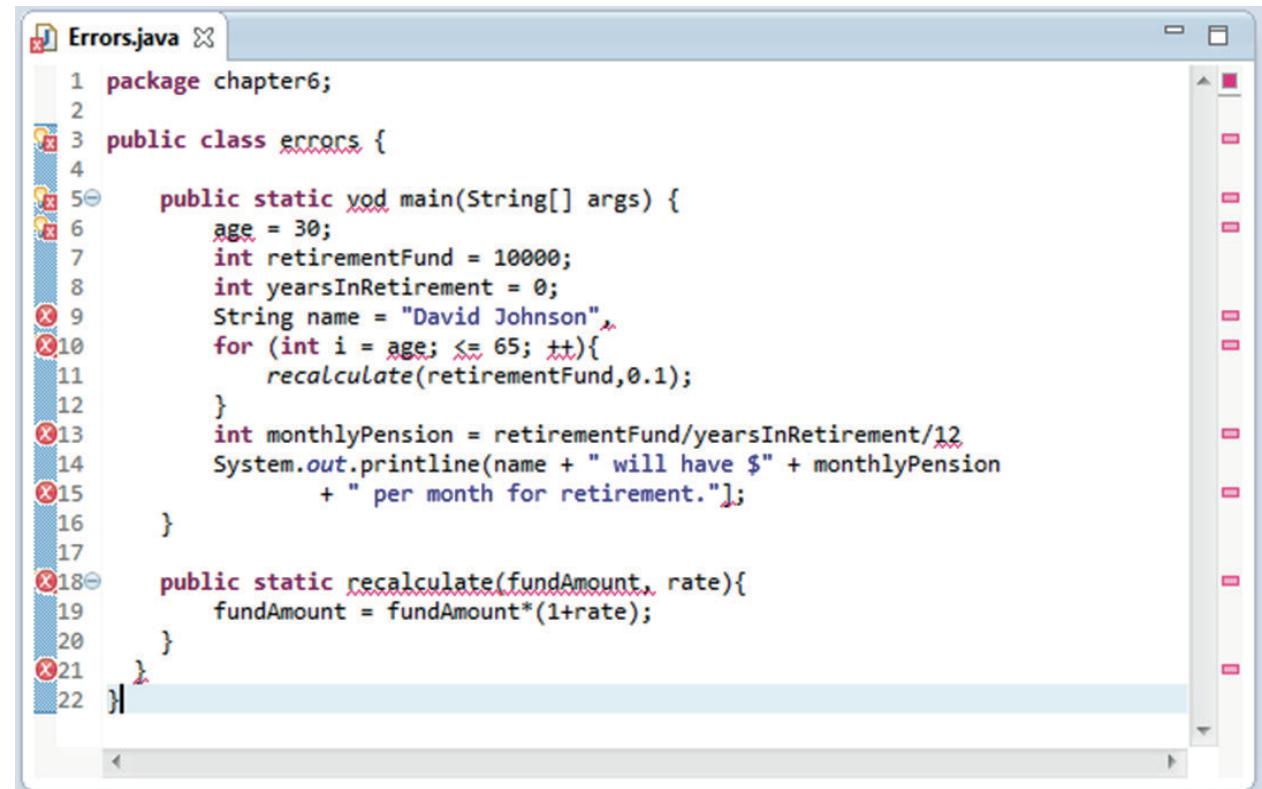
Ligne 18 : Ajoute les types de données des paramètres à double fundAmount, double rate

Ligne 13 : Ajoutez un point-virgule à la fin.

Ligne 14 : remplacez par println (println n'est pas une syntaxe correcte).

Ligne 15 : Remplacez le crochet par une parenthèse.

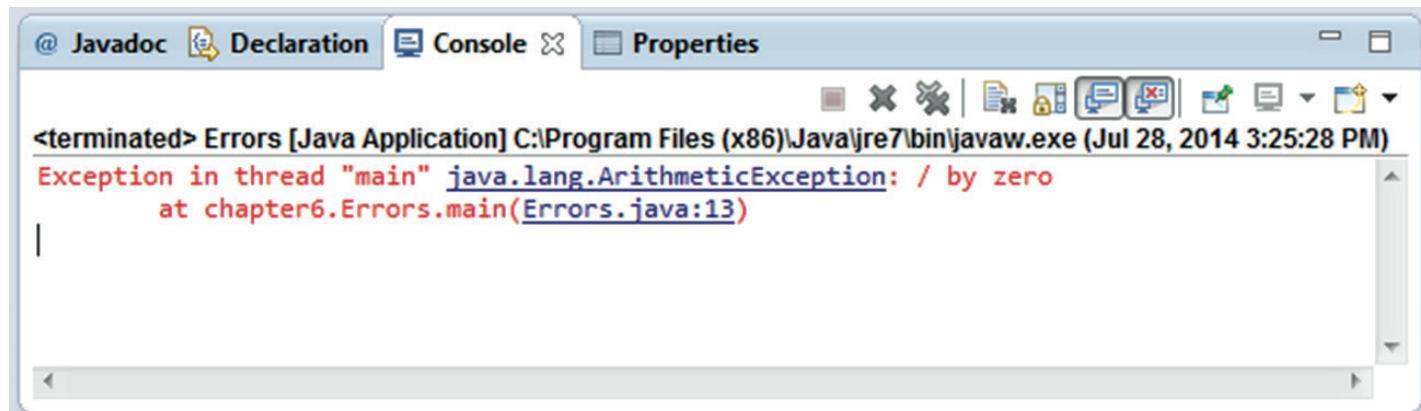
Ligne 22 : Supprimez la dernière accolade fermante (il y en avait une de trop).



A screenshot of a Java code editor showing the file Errors.java. The code contains several syntax errors, each marked with a red 'X' icon next to the line number. The errors are:

- Ligne 3: public class errors { (missing 'e' in 'errors')
- Ligne 5: public static yod main(String[] args) { (misspelled 'void')
- Ligne 9: age = 30; (missing semicolon)
- Ligne 10: int retirementFund = 10000; (missing semicolon)
- Ligne 11: int yearsInRetirement = 0; (missing semicolon)
- Ligne 12: String name = "David Johnson", (extra comma)
- Ligne 13: for (int i = age; <= 65; <=){ (extra '<' and extra '}' at the end of the loop)
- Ligne 14: calculate(retirementFund, 0.1); (misspelled 'recalculate')
- Ligne 15: int monthlyPension = retirementFund/yearsInRetirement/12; (extra '/' at the end of the division)
- Ligne 16: System.out.println(name + " will have \$" + monthlyPension + " per month for retirement."); (extra '+' before 'per month for retirement.')
- Ligne 18: public static recalculat(fundAmount, rate){ (misspelled 'recalculate')
- Ligne 19: fundAmount = fundAmount*(1+rate); (extra 'f' in 'fundAmount')
- Ligne 21: } (extra closing brace at the end of the class definition)

- Si vous avez essayé d'exécuter le programme après avoir repéré toutes les erreurs de syntaxe, vous avez probablement déjà trouvé votre première erreur d'exécution. Dans la console, vous verrez un texte rouge indiquant qu'il y a eu une exception dans le thread main.
- L'exception est une classe Java comprenant de nombreux types de problèmes d'exécution.
- Java fait la distinction entre les exceptions et les erreurs : les exceptions peuvent - et doivent - être gérées par le programmeur, tandis que les erreurs sont des problèmes graves que les programmes raisonnables ne sont pas censés gérer.

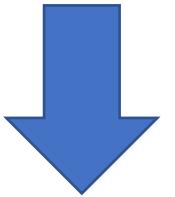


- Si vous allez à la ligne 13, vous verrez que le programme essaie de diviser retirementFund par yearsInRetirement.
- Si vous reportez aux initialisations de ces deux variables, vous verrez que retirementFund = 10000 et yearsInRetirement = 0.
- C'est là que réside le problème ! Si vous fixez la valeur de yearsInRetirement à une autre valeur, disons 20 ans, pour une personne prenant sa retraite à 65 ans et vivant jusqu'à 85 ans, vous n'aurez plus d'exception de division par zéro.

- Un programme fiable produit des résultats corrects pour toutes les entrées valides.
- Il ne s'agit pas d'un programme fiable s'il ne produit des résultats corrects que pour certaines valeurs d'entrée.
- Un autre critère important de la fiabilité d'un programme est la robustesse, qui mesure le bon fonctionnement du programme dans diverses conditions. Si un programme se bloque trop facilement lorsqu'un mauvais type d'argument est transmis à une méthode ou qu'une valeur d'entrée invalide est saisie, on ne peut pas dire que le programme soit très robuste.
- Le traitement des **exceptions** peut être utilisé pour améliorer la **robustesse** du programme.

- Jusqu'à présent, nous avons laissé le système gérer les exceptions levées.
- Cependant, lorsque nous laissons le système gérer les exceptions, une seule exception lancée aura très probablement pour conséquence des résultats erronés ou l'arrêt du programme.
- Au lieu de dépendre du système pour la gestion des exceptions, nous pouvons augmenter la fiabilité et la robustesse du programme si nous attrapons nous-mêmes les exceptions en incluant des routines de récupération d'erreur dans notre programme.

```
Scanner scanner = new Scanner(System.in);
System.out.print("Enter integer: ");
int number = scanner.nextInt();
```



abc123

```
Exception in thread "main" java.util.InputMismatchException
at java.util.Scanner.throwFor(Scanner.java:819)
at java.util.Scanner.next(Scanner.java:1431)
at java.util.Scanner.nextInt(Scanner.java:2040)
at java.util.Scanner.nextInt(Scanner.java:2000)
at Ch8Sample1.main(Ch8Sample1.java:35)
```

Ce message d'erreur indique que le système a détecté une exception appelée Input- MismatchException, une erreur qui se produit lorsque nous essayons de convertir une chaîne de caractères qui ne peut pas être convertie en une valeur numérique.

En utilisant la classe de service **AgeInputVer1**, on écrit un programme qui obtient l'âge d'une personne et répond avec l'année de naissance de la personne.

Notez que le programme tient compte du fait que la personne n'a pas déjà eu son anniversaire cette année.

```
import java.util.*;
class AgeInputVer1 {
    private static final String DEFAULT_MESSAGE = "Your age: ";
    private Scanner scanner;
    public AgeInputVer1() {
        scanner = new Scanner(System.in);
    }
    public int getAge() {
        return getAge(DEFAULT_MESSAGE);
    }
    public int getAge(String prompt) {
        System.out.print(prompt);
        int age = scanner.nextInt();
        return age;
    }
}
import java.util.*;
class Ch8AgeInputMain {
    public static void main(String[] args) {
        GregorianCalendar today;
        int age, thisYear, bornYr;
        String answer;
        Scanner scanner = new Scanner(System.in);
        AgeInputVer1 input = new AgeInputVer1();
        age = input.getAge("How old are you? ");
        today = new GregorianCalendar();
        thisYear = today.get(Calendar.YEAR);
        bornYr = thisYear - age;
        System.out.print("Already had your birthday this year? (Y or N)");
        answer = scanner.next();
        if (answer.equals("N") || answer.equals("n")) {
            bornYr--;
        }
        System.out.println("\nYou are born in " + bornYr);
    }
}
```

Le programme fonctionne bien tant qu'une entrée valide est saisie. Mais que se passe-t-il si l'utilisateur épelle l'âge, disons neuf au lieu de 9 ? Une exception input-mismatch est levée parce que la valeur d'entrée neuf ne peut pas être convertie en un nombre entier en utilisant la méthode parseInt. Dans l'implémentation actuelle, le système traite l'exception levée en affichant le message d'erreur suivant et en mettant fin au programme.

```
import java.util.*;
class AgeInputVer1 {
    private static final String DEFAULT_MESSAGE = "Your age: ";
    private Scanner scanner;
    public AgeInputVer1() {
        scanner = new Scanner(System.in);
    }
    public int getAge() {
        return getAge(DEFAULT_MESSAGE);
    }
    public int getAge(String prompt) {
        System.out.print(prompt);
        int age = scanner.nextInt();
        return age;
    }
}
```

```
Exception in thread "main" java.util.InputMismatchException
at java.util.Scanner.throwFor(Scanner.java:819)
at java.util.Scanner.next(Scanner.java:1431)
at java.util.Scanner.nextInt(Scanner.java:2040)
at java.util.Scanner.nextInt(Scanner.java:2000)
at AgeInputVer1.getAge(AgeInputVer1.java:48)
at Ch8AgeInputMain.main(Ch8AgeInputMain.java:30)
```

```
import java.util.*;
class Ch8AgeInputMain {
    public static void main(String[] args) {
        GregorianCalendar today;
        int age, thisYear, bornYr;
        String answer;
        Scanner scanner = new Scanner(System.in);
        AgeInputVer1 input = new AgeInputVer1();
        age = input.getAge("How old are you? ");
        today = new GregorianCalendar();
        thisYear = today.get(Calendar.YEAR);
        bornYr = thisYear - age;
        System.out.print("Already had your birthday this year? (Y or N)");
        answer = scanner.next();
        if (answer.equals("N") || answer.equals("n")) {
            bornYr--;
        }
        System.out.println("\nYou are born in " + bornYr);
    }
}
```

Une meilleure alternative serait de gérer nous-mêmes l'exception levée.

Modifions la méthode getAge afin qu'elle tourne en boucle jusqu'à ce qu'une entrée valide pouvant être convertie en un nombre entier soit saisie.

⇒ envelopper les instructions qui peuvent potentiellement lever une exception avec l'instruction de contrôle try-catch.

⇒ Dans cet exemple, il n'y a qu'une seule instruction qui peut potentiellement lever une exception, à savoir: **age = scanner.nextInt();**

Nous plaçons cette instruction dans le bloc try et les instructions que nous voulons voir exécutées en réponse à l'exception levée dans le bloc catch correspondant.

Les instructions du bloc try sont exécutées en séquence. Lorsque l'une des instructions lève une exception, le contrôle est transmis au bloc catch correspondant et les instructions à l'intérieur du bloc catch sont exécutées. L'exécution se poursuit ensuite avec l'instruction qui suit cette instruction du bloc d'essai, en ignorant toutes les autres instructions du bloc d'essai. Si aucune instruction du bloc try ne lève d'exception, le bloc catch est ignoré et l'exécution se poursuit avec l'instruction qui suit cette instruction try-catch.

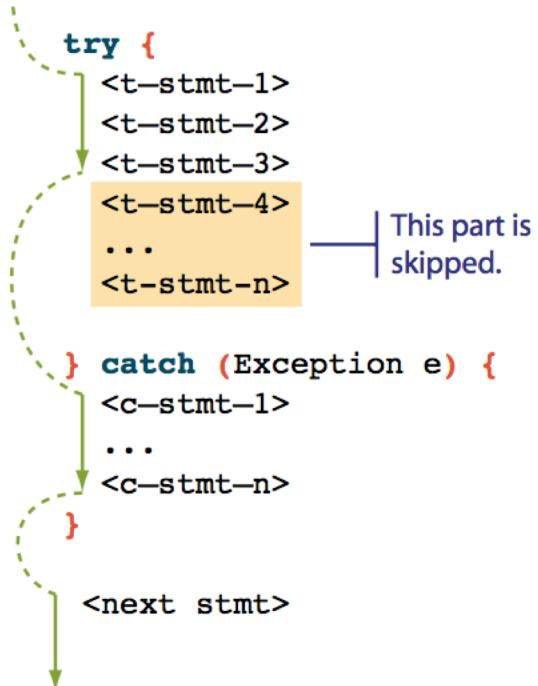
```
try {
    age = scanner.nextInt();
} catch (InputMismatchException e) {
    System.out.println(
        "Invalid Entry. Please enter digits only.");
}
```

A statement that could throw an exception

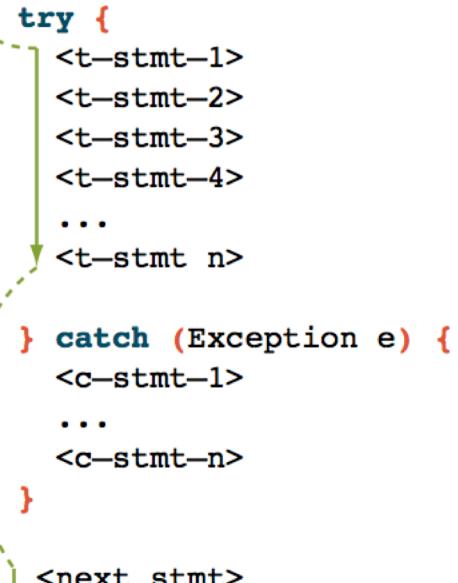
The type of exception to be caught

Exception

Assume **<t-stmt-3>** throws an exception.



No exception



- Cette version est plus robuste car le programme ne s'arrête pas brusquement lorsqu'une valeur non valide est saisie.
- Il est encore possible de l'améliorer. Par exemple, l'implémentation actuelle accepte les entiers négatifs invalides. Puisque l'âge négatif n'est pas possible, améliorons le code en interdisant l'entrée d'entiers négatifs. Remarquez qu'un nombre entier négatif est un nombre entier, donc la méthode nextInt ne lèvera pas d'exception.

```
while (true) {
    System.out.print(prompt);

    try {
        age = scanner.nextInt();

        if (age < 0) {
            throw new Exception("Negative age is invalid");
        }

        return age; //input okay so return the value & exit
    } catch (InputMismatchException e) {

        scanner.next();

        System.out.println("Input is invalid.\n" +
                           "Please enter digits only");

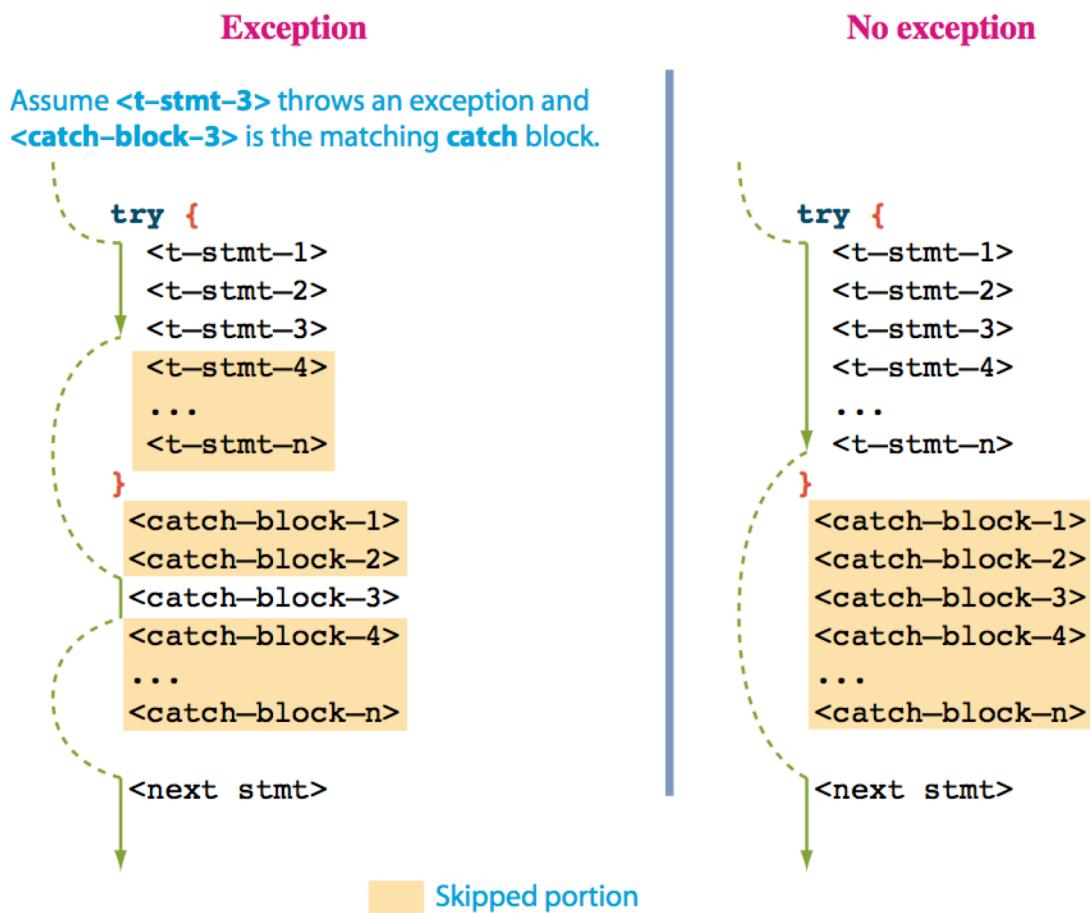
    } catch (Exception e) {

        System.out.println("Error: " + e.getMessage());
    }
}
```

Throws an exception when age is a negative integer.

The thrown exception is caught by this catch block.

- Multiple catch blocks



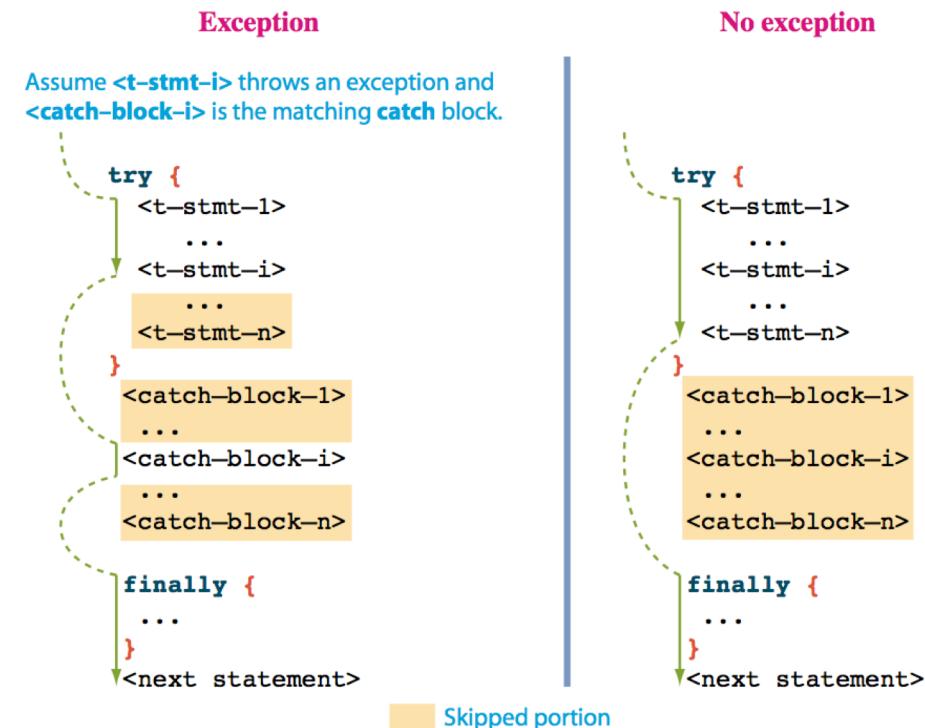
- S'il y a un bloc de code qui doit être exécuté, qu'une exception soit levée ou non, nous utilisons le mot réservé finally.

```

try {
    num = scanner.nextInt();

    if (num > 100) {
        throw new Exception("Out of bound");
    }
} catch (InputMismatchException e) {
    scanner.next();
    System.out.println("Not an integer");
} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
} finally {
    System.out.println("DONE");
}

```



- Le mécanisme de traitement des exceptions peut être considéré comme une autre forme de structure de contrôle.
- Une exception représente une condition d'erreur qui peut se produire au cours de l'exécution normale du programme. Lorsqu'une exception se produit, la séquence normale d'exécution est interrompue et la routine de gestion des exceptions est exécutée.
- Lorsqu'une exception se produit, on dit qu'une exception est lancée (an exception is thrown).
- Lorsque le code de gestion des exceptions correspondant est exécuté, on dit que l'exception lancée est attrapée (an exception is caught).
- En utilisant judicieusement les routines de gestion des exceptions dans notre code, nous pouvons en augmenter la robustesse.