Résumé Introduction Programmation Java

Concepts

Un programme : séquence, test conditionnel, boucles.

Objets : Les objets Java modélisent les objets d'un problème donné

Classe : Les objets sont crées à partir de classes. La classe décrit le type d'objet ; Une classe est composée de champs, constructeurs, méthodes.

Instance : Un objet est une instance d'une classe, il est possible de créer plusieurs objets similaires à partir d'une classe unique.

Méthode (methods) : Il est possible de communiquer avec des objets en invoquant des méthodes. En-tête et un corps

Signature : L'en-tête d'une méthode s'appelle une signature. Il apporte les informations nécessaires à l'appel de cette méthode.

Paramètre : Les méthodes peuvent disposer de paramètres qui leur permettent de fournir des informations supplémentaires pour une tâche déterminée.

Type : Les paramètres possèdent un type. Le type définit les catégories de valeurs qui peuvent être associées à un paramètre.

État : Les objets disposent d'un état représenté par les valeurs stockées dans les champs.

Champs (fields) : Les champs stockent des données qui seront utilisées par un objet. Ils sont aussi appelés variables d'instance.

Constructeur (constructors) : Les constructeurs permettent de configurer correctement chaque objet lors de sa création.

Variables : portée, durée de vie, local ; les variables de type objet stockent des références aux objets

= instruction d'affectation : stockent la valeur présente du côté droite de l'instruction dans la variable de gauche.

Méthode d'accès (accessor methods) : renvoie des informations sur l'état d'un objet

Méthodes de modification (mutator methods) : modifient l'état d'un objet.

Abstraction : capacité d'ignorer les détails pour se concentrer sur un niveau supérieur du problème.

Modularité : processus de division d'un tout en parties bien définies, qui peuvent être élaborées et étudiées séparément, et qui interagissent de façons bien définies.

Diagramme de classes : les classes d'une application et leurs relations.

Diagramme d'objets : montre les objets et leurs relations à un moment donné au cours de l'exécution d'une application.

Types primitifs : types qui ne sont pas des objets : *int, boolean, char, double, long*, ils n'ont pas de méthodes.

Surcharges : méthode ou constructeur avec même nom, mais paramètres différents

Appel de méthode externe : *Objet.nomMéthode* ou soi-même avec *this*

Collection (ArrayList) : les objets collections sont des objets qui peuvent stocker des quantités arbitraires d'autres objets.

Itérateur (iterator) : objet qui permet de parcourir tous les éléments d'une collection.

Tableau ([]) : type spécial de collection capable de stocker un nombre déterminé d'éléments.

Interface d'une classe : décrit ce que fait la classe et la manière de l'utiliser sans montrer l'implantation.

Implantation : code source d'une classe.

Objet inaltérables : quand il ne peut être modifié une fois crée, exemple string.

Association (HashMap) : collection qui stocke des paires clé/valeur sous forme d'entrées.

Ensemble (HashSet) : collection qui mémorise chaque élément au maximum une fois. Il n'y a pas d'ordre entre les éléments d'un ensemble.

Modificateurs d'accès (private, public, protected) : définissent la visibilité d'un champ, d'un constructeur ou d'une méthode. Les éléments publics sont accessibles de l'intérieur de la même classe et à partir des autres classes ; les éléments privés ne sont disponibles qu'à partir de l'intérieur de la même classe : les éléments protected donnent accès aux sous-classes. Sans rien visibilité = package.

Variable de classe (static) : un seul exemplaire d'une variable de classe peut exister à un instant T, indépendamment du nombre d'instances créées.

Constantes (final)

Méthode static : appel à la class et pas à l'objet, donc pas besoin que l'objet soit instancié.

Cohésion : mesure la qualité du recouvrement entre une partie du code et la tâche logique à remplir. Dans un système très cohérent, chaque partie de code (méthode, classe ou module) est responsable d'une tâche bien identifiée. Une bonne conception de classes présente un haut degré de cohésion. => reusable.

Une et une seul tâche bien définie (méthodes), un rôle défini (classe).

Responsability driven design : owner of data = processor of data; refactoring, design guidelines

Couplage : faible (loose), possibilité de changer la class sans affecter les autres

Compiler, instruction set, virtual machine = interpreters, bytecode

Stack (pile) : LIFO, used for nested method calls (méthode imbriqué)

Heap (tas) : any order, grabage collector cleans up

Recursive algorithms

Héritage : permet de définir une classe comme extension d'une autre

Superclasse : est une classe qui est étendue par une autre classe

Sous-classe : une classe qui étend (hérite de) une autre classe. (champs et méthodes)

Constructeur de superclass : doit être appelé par le constructeur de la sous classe avec super()

Sous-typage : substitution, polymorphisme, super type Objet

Type statique : type de la variable tel qu'il est déclaré dans le code source de la classe.

Type dynamique : type de l'objet qui est actuellement stocké dans la variable.

Surcharge (overloading) & Redéfinition (overriding)

Classe abstraite : ne peut pas être instancié, contient des méthodes normal et abstraite

Méthode abstraite : signature de méthode uniquement, sans corps.

Interface : permet l'héritage multiple ; ne définit pas d'implantation pour les méthodes.

Vérifier les arguments.

Traiter des erreurs avec des valeurs de retours ne garanti pas le traitement => besoin d'exception.

Exception : exceptions sous contrôle vérifié par le compilateur, il faut utiliser throws et try

RuntimeException : exceptions hors contrôle emploi n'est pas vérifié par le compilateur

Code

Création d'objet : *new* Name()

Classe :

```
public class ClassName
{
        Champs
        Constructeurs
        Méthodes
}
```

Constructeur : *public* ClassName()

Acceseurs : *public int* getmethodeName()

Modificateur : *public void* setmethodeName()

Affichage terminal : *System.out.println() ;*

Instruction conditionnelle :

```
if(test) {

} else {

}
```

Variable local: type name **Attention initialisation**

Champs: privat type name **Attention initialisation**

Opérateurs logiques :

&& (et) court-circuit (2ème partie pas exécuté)
|| (ou) court-circuit (2ème partie pas exécuté)
^(ou exclusif)
! (non)

+ (concaténation)
% (modulo)
null
i++ post incrémentation
++i pré incrémentation

Attention, comparer les chaînes avec .equals et pas avec == car référence à l'objet et non au contenu.

Boucle :

```
while(condition de la boucle) {
        corps de la boucle
}
```

```
for(initialization (int i =0); condition; action après le corps) {
        bloc à répéter
}
```

Iterator : objet.*iterator(),* .it.*next(),* it.*hasNext()*

Transtypage : value = (type) value

Array : type[] variablename; new type[xx]

Random : .*nextInt(*size) (utiliser un seul)

Namespace : *import java.util.* ;*

ArrayList : *add, size, remove, get* 0-(n-1)

HashMap : .*put,(*key, value*) .get(*key*)*

HashSet : .*add*(value)

String arg[] = contiens tous les arguments

abstract public void test(); Attention ;

C1 extends C2 : true

C1 extends C2, C3 : false

I1 extends I2, I3 : true

C2 extends I1 : false

I1 extends C1 : false

C1 implements I1, I2 : true

I1 implements C1 : false

extends = étendre toujours le même type

implements = toujours une interface

Lancer une erreur :

```
throw new IllegalArgumentException("text");
```

Récupération sur erreur :

```
boolean successful = false;
int attempts = 0;
do {
        try{
        successful = true;
        }
        catch(Exception e){

        }finally{ //toujours executé }
} while (!successful && attempts < MAX);
```

Java 5

Generics :

```
private HashMap<String, String> responses ;
responses = new HashMap<String, String> ();

Iterator<Lot> it = lots.iterator();
```

Foreach :

```
for(Type element : array){
        element
}
```

Enumerated Type :

```
Public enum Level {
        LOW, MEDIUM, HIGH
}
```

Scanner :

```
Scanner tokenizer = new Scanner(line);
for(int i = 0; i < dataline.length; i++){
        dataline[i] = tokenizer.nextInt();
}
```

Autoboxing & Unboxing

Static Imports