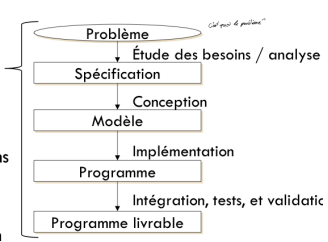
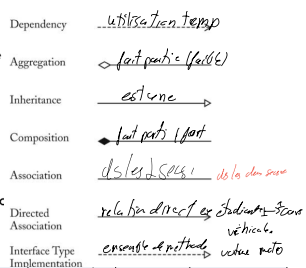
Cycle traditionnel de développement (en cascades)

- Nécessité de connaître parfaitement les besoins
- Résultats vérifiés tard dans le cycle
- Une faute détectée tard peut avoir un impact et un coût élevé





Tag & Parameter	Usage	Applies to
@author *John Smith*	Describes an author.	C, I, E
@version *version*	Provides software version entry. Max one per Class or Interface.	C, I, E
@since *since-text*	Describes when this functionality has first existed.	C, I, E, F, M
@see *reference*	Provides a link to other element of documentation.	C, I, E, F, M
@param *name description*	Describes a method parameter.	M
@return *description*	Describes the return value.	M
@exception *classname description* **@throws** *classname description*	Describes an exception that may be thrown from this method.	M
@deprecated *description*	Describes an outdated method.	C, I, E, F, M
{**@inheritDoc**}	Copies the description from the overridden method.	OM
{**@link** *reference*}	Link to other symbol.	C, I, E, F, M

Cycle de développement itératif et incrémental

- Itération
  - Mini-projet à résultat testé et exécutable
  - Feedback rapide et minimisation des risques
- Incréments
  - Le système croît après chaque itération
  - Convergence vers le produit final

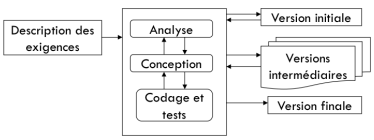
abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Visibilité	Classes dans le même *package*	Ses sous-classes	Autres classes
public	Oui	Oui	Oui
protected	Oui	Oui	Non
private	Non	Non	Non

C: Class, I: Interface,
E: Enum, F: Field
M: Method,
OM: Overriding Method

- **Variable final** → pour créer une variable constante
- **Méthode final** → pour empêcher la redéfinition
- **Classe final** → pour empêcher l'héritage

Cycle de développement itératif et incrémental




```
class ClasseMuable {
  private Date d;

  public  ClasseMuable() {
    d = new Date();
  }
  public Date getDate() {
    return d;
  }
}
```

```
class Main {
  private Date d;

  public static void main(String[] args) {
    ClasseMuable cm = new ClasseMuable();
    d = new Date();
    d = cm.getDate();
    d.setDate(…)
  }
}
```

Quand une sous-classe a une méthode avec la même **signature** (*method overriding*)

- On peut appeler la méthode redéfinie avec **super** :

```
class Triangle extends Figure {
  @Override
  public void afficher() {
    super.afficher();
    System.out.println("Nom : " + nom);
  }
```

```
public double getSalary() {
  return getSalary() + bonus; //  ERREUR : appel récursi
}
```

Il faut utiliser le mot réservé « super » :

```
public double getSalary() {
  return super.getSalary() + bonus;
}
```

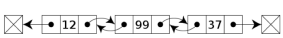
```
public class institution{

}

public class universite extends institution{
      public universite() {
            super();
      }
}
```

Utiliser le mot réservé « *super* » dans le constructeur de la sous-classe

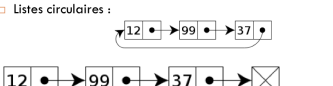
```
public Manager(String aName) {
  super(aName); // Appelle le constructeur de la superclasse pour initialiser ses champs privés
  bonus = 0;
}
```

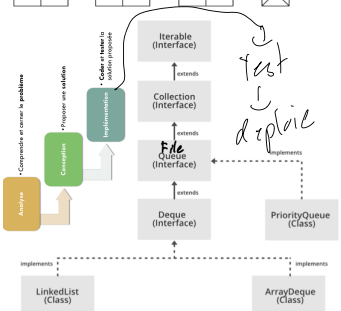
```
List<String> liste = new LinkedList<String>();
liste.add("Bonjour");
liste.add("Monde");
liste.add("Mercredi");
liste.remove(1);
System.out.println("Taille " + liste.size( ));
String deuxieme = liste.get(1);
System.out.println(" Deuxieme " + deuxieme);
```

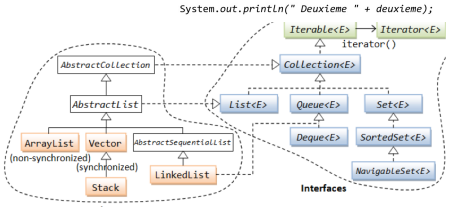
- Listes doublement chaînées :

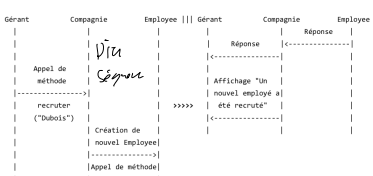


- Listes circulaires :









Dessinez le diagramme de séquence correspondant à l'appel de la méthode gérant.recruter("Dubois") dans le code suivant :

```
public class Gerant {
private Compagnie maCompagnie = new Compagnie("ma compagnie");
public void recruter(String nom){
maCompagnie.addEmployee(nouveau);
System.out.println("Un nouvel employé a été recruté");
```

Une **cabine téléphonique** contient un **téléphone** composé de **9 touches** pouvant être **enfoncées** par l'**utilisateur**. Le téléphone peut être utilisé par un utilisateur, lequel doit réaliser plusieurs opérations avec le dispositif : le **décrocher, tenter une connexion** avec un **numéro**, et **communiquer** un **message** si la personne répond.

- Relation d'agrégation (utilisation) : La classe CabineTéléphonique contient un téléphone (Téléphone). Cela signifie que CabineTéléphonique utilise un téléphone pour fournir ses fonctionnalités.

Téléphone - Touche :

- Relation de composition : La classe Téléphone est composée de neuf touches (Touche). Cela signifie que les touches sont des composants essentiels du téléphone, et lorsque le téléphone est détruit, les touches sont également.

CabineTéléphonique - Utilisateur :

- Relation d'association : La classe CabineTéléphonique peut être utilisée par un utilisateur (Utilisateur). Cela signifie qu'un utilisateur peut interagir avec la cabine téléphonique en utilisant le téléphone.

Touche - CabineTéléphonique (optionnel, selon les besoins) :

- Relation d'association : Chaque touche pourrait potentiellement être associée à une cabine téléphonique pour indiquer à quelle cabine elle est connectée, si cela est nécessaire pour votre scénario.

Utilisateur – Téléphone (Association)

- L'utilisateur est associé au téléphone lorsqu'il utilise la cabine téléphonique. L'utilisateur utilise le téléphone pour décrocher, composer un numéro et communiquer un message. Le téléphone est l'interface par laquelle l'utilisateur effectue ces actions.

Multiplicité :

- Un utilisateur est généralement associé à un téléphone à la fois lors de son utilisation de la cabine téléphonique.
- Chaque téléphone est utilisé par un utilisateur à un moment donné.

1. **Classe : CabineTéléphonique** Diagramme classe
   - Attributs :
     - **téléphone: Téléphone** (une instance du classe Téléphone).
   - Méthodes :
     - Aucune méthode.
2. **Classe : Téléphone**
   - Attributs :
     - **touches: Touche[]** (un tableau de 9 touches)
   - Méthodes :
     - Aucune méthode n'est spécifiée pour cette classe.
3. **Classe : Touche**
   - Attributs :
     - **valeur: string** (pour représenter la valeur/le numéro associé à la touche
   - Méthodes :
     - **enfoncer()**
4. **Classe : Utilisateur**
   - Attributs :
     - Aucun attribut n'est spécifié pour cette classe.
   - Méthodes :
     - **décrocher()**
     - **composerNumero(numero: string)**
     - **communiquer(message: string)**

```
public class ListeDoublementChainee {
  // Classe interne pour représenter un nœud
  class Noeud {
    Object valeur;
    Noeud precedent, suivant;
    public Noeud(Object valeur) {
      this.valeur = valeur;
    }
  }
  private Noeud tete, queue; // Tête et queue de la liste
  private int taille; // Taille de la liste
  // Ajouter un élément à la fin
  public void ajouter(Object valeur) {
    Noeud nouveau = new Noeud(valeur);
    if (tete == null) tete = queue = nouveau; // Liste vide
    else {
      queue.suivant = nouveau;
      nouveau.precedent = queue;
      queue = nouveau; // Mettre à jour la queue
    }
    taille++;
  }
  // Supprimer un élément par sa valeur
  public void supprimer(Object valeur) {
    Noeud courant = tete;
    while (courant != null) {
      if (courant.valeur.equals(valeur)) {
        if (courant.precedent != null)
          courant.precedent.suivant = courant.suivant;
        else tete = courant.suivant;
        if (courant.suivant != null)
          courant.suivant.precedent = courant.precedent;
        else queue = courant.precedent;
        taille--;
        return;
      }
      courant = courant.suivant;
    }
  }
  // Afficher la liste
  public void afficher() {
    Noeud courant = tete;
    while (courant != null) {
      System.out.print(courant.valeur + " <-> ");
      courant = courant.suivant;
    }
    System.out.println("null");
  }
  // Méthode principale pour tester la liste
  public static void main(String[] args) {
    ListeDoublementChainee liste = new ListeDoublementChainee();
    liste.ajouter("A");
    liste.ajouter("B");
    liste.ajouter("C");
    liste.afficher(); // Sortie : A <-> B <-> C <-> null
    liste.supprimer("B");
    liste.afficher(); // Sortie : A <-> C <-> null
  }
}
```

Cycle de vie
Ensemble des phases par lesquelles passe un logiciel durant sa durée de vie (conception, mise en œuvre, maintenance et mise hors-service)
Cycle (ou processus) de développement
Ensemble des étapes menant à la mise en œuvre d'un logiciel
Il faut utiliser un processus bien défini
Des étapes et des livrables identifiés
Cycle de développement itératif et incrémental
Itération
Mini-projet à résultat testé et exécutable
Feedback rapide et minimisation des risques
Incréments
Le système croît après chaque itération
Convergence vers le produit final

Classe : usine qui permet de créer des objets
Décrit les attributs représentant l'état
Définit les méthodes pour le comportement

Un objet créé à partir d'une classe est une instance de cette classe
?Student bob = new Student();
Un objet est défini par :
Son état
Décrit par l'ensemble des propriétés de l'objet
Il peut changer à travers le temps
Son comportement
Défini par l'ensemble des opérations qu'il supporte
Dépend de son état
Son identité

Définit un type (abstraction)
Définit les propriétés / attributs de ses objets et les valeurs pouvant être affectées à ces attributs
Définit les opérations que ses objets peuvent réaliser
Décrit la façon d'interagir avec ses objets
Une classe spécifie le comportement et les états possibles d'un ensemble d'objets de même type
Définit les états possibles de ses objets

La classe Object est la racine de la hiérarchie des classes ? Toutes les classes héritent de la classe Object par défaut
? La classe Object définit plusieurs méthodes qui sont souvent redéfinies
? La méthode pour afficher l'information d'un objet (appelée avec System.out.print) :
public String toString()
? La méthode pour évaluer si deux objets sont égaux ou non :
public boolean equals(Object obj)

Si le constructeur de la sous-classe n'appelle pas « super », la super-
classe doit avoir un constructeur sans paramètre, qui sera alors
appelé (qui est automatiquement déclaré si l'utilisateur ne l'a pas
déclaré).

Toutes les exceptions doivent être attrapées à un point dans l'exécution
du code
? Si on n'attrape pas l'exception dans la méthode où elle est lancée,
cette méthode passe la responsabilité à la méthode qui l'a appelée
? On utilise le mot-clé throws pour exprimer cette responsabilité
? Si nécessaire, on répète ce processus (traverse la pile d'appels de
méthodes) jusqu'au début de l'exécution (méthode main)
? Si le compilateur ne trouve aucun code qui attrape une exception, il
signale une erreur comme :
ExceptionExample.java:43: unreported exception java.lang.Exception; must be caught

Sert à évaluer le temps de calcul et/ou l'espace mémoire requis pour
dérouler un algorithme
? On peut ainsi comparer les algorithmes ? S'exprime en fonction du nombre de données et de leur taille
? S'intéresse seulement à l'ordre de grandeur
■ Pour n données :
■ Complexité logarithmique : O(log(n)) ■ Complexité polynomiale : O(n) linéaire, O(n2) quadratique, etc. ■ Complexité exponentielle : O(xn)
? Toutes les opérations sont considérées équivalentes ? En général, on ne considère que les pires cas
? Sert aussi à évaluer la difficulté d'un problème et à le classifier
? Problèmes NP-difficiles (NP-hard), NP-complets (NP-complete), etc.

Collection : interface très générale ? Structures de données abstraites :
? Set : chaque élément de la collection est unique
? SortedSet : les éléments sont triés selon un ordre donné
? List : une collection ordonnée (indexée) ? Structures de données concrètes :
? HashSet : une implémentation de Set qui utilise le hachage pour localiser les éléments du Set ? TreeSet : une implémentation de SortedSet qui stocke ses
éléments dans un arbre binaire équilibré
? LinkedList, ArrayList, Vector : quelques implémentations de l'interface List

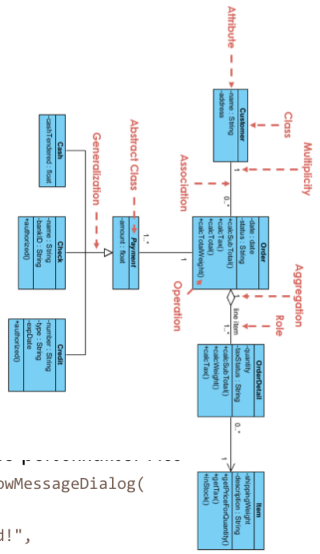
Structure de données – manière particulière d'organiser les données pour une utilisation efficace
? Bâties sur les structures de données primitives
? S'inspirent de structures mathématiques connues
? S'accompagnent, en POO, d'opérations essentielles ? Quelques structures de données :
? Tableaux et listes
? Tableaux associatifs, tables de hachage
? Ensembles
? Arbres et graphes
? Classes et objets

Groupe de nœuds qui, pris ensemble, forment une séquence
? Sous la forme la plus simple, chaque nœud est composée d'une donnée
et d'une référence (un lien) vers le nœud suivant dans la séquence
? Structure efficace pour l'ajout et la suppression d'éléments à n'importe quelle position de la séquence ? Permettent d'implémenter des types abstraits tels que :
? Les piles (stack)
? Les files (queue)

Constructeurs :
? LinkedList() – celui-ci construit une liste vide ? LinkedList(Collection<? extends E> c) – celui-ci construit une
liste basée sur une autre collection
? Quelques méthodes importantes :
? boolean add(E e) – ajoute un élément à la fin de la liste
? E get(int index) – retourne un élément à la position index, le
premier élément se retrouve à la position 0 ? E remove(int index) – efface l'élément à la position index

TreeSet<S> – constructeurs et méthodes importants
? Constructeurs :
éléments du type E ou ? TreeSet() – celui-ci construit un ensemble vide des sous-classes de E
? TreeSet(SortedSet<E> s) – celui-ci construit un ensemble basé sur un autre ensemble ? TreeSet(Collection<? extends E> c) – celui-ci construit un ensemble basé sur un autre
collection ? TreeSet(Comparator<? super E> comparator) – celui-ci construit un ensemble vide
qui va être ordonné avec un comparateur spécial
objet qui implémente l'interface Comparator pour un type E ou pour une des super-classes de E
les éléments entre fromElement (inclusif) et toElement (exclusif), i.e. [fromElement. toElement)
? Quelques méthodes importantes :
? E first() – retourne le premier élément (le plus petit) dans l'ensemble
? E last() – retourne le dernier élément (le plus grand) dans l'ensemble
? SortedSet<E> subSet(E fromElement, E toElement) – retourne un sous-ensemble

```
import java.util.Scanner;
public class ManipulationDeChaine {
    public static String chaine = "";
    public static boolean
estPalindrome(String chaine){
        String temp = "";
        for(int i = chaine.length()-1; i >=0;
i--){
            temp += chaine.charAt(i);
        }
        return temp.equals(chaine);
    }
    public static void main(String[] args)
{
        Scanner sc =  new
Scanner(System.in);
        chaine =sc.nextLine();
        ManipulationDeChaine mdc = new
ManipulationDeChaine();
        System.out.println("Est-ce que \""
+ chaine + "\" est un palindrome ? " +
mdc.estPalindrome(chaine));
    }
```




```
JOptionPane.showMessageDialog(
  null,
  "Hello, World!",
  "Welcome Message",
JOptionPane.INFORMATION_MESSAGE, /
  new ImageIcon("earth-icon.png")
);
```

```
public class Student {
    private int id;
    private String name;
    private double[] grades = new double[3];

    // Constructor
    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // Add grades
    public void addGrades(double grade1, double grade2, double grade3) {
        grades[0] = grade1;
        grades[1] = grade2;
        grades[2] = grade3;
    }

    // Calculate average
    public double calcAvg() {
        double sum = 0.0;
        for (double grade : grades) sum += grade;
        return sum / grades.length;
    }

    // Display student info
    public void displayInfo() {
        System.out.println("ID: " + id + ", Name: " + name + ", Average: " + calcAvg());
    }

    public static void main(String[] args) {
        Student student = new Student(1, "Alice");
        student.addGrades(85.5, 90.0, 78.0);
        student.displayInfo();
    }
}
```

```
public class Student {
    private double[] grades = new double[3]; // Tableau de 3 notes
    // Méthode pour ajouter une note
    public void setGrade(int index, double grade) throws
IllegalArgumentException {
        if (index < 0 || index >= grades.length) {
            throw new IllegalArgumentException("Index invalide : " + index);
        }
        if (grade < 0 || grade > 100) {
            throw new IllegalArgumentException("La note doit être comprise
entre 0 et 100.");
        }
        grades[index] = grade;
    }
    // Méthode pour calculer la moyenne
    public double calcAvg() throws ArithmeticException {
        double sum = 0.0;
        for (double grade : grades) {
            sum += grade;
        }
        if (grades.length == 0) {
            throw new ArithmeticException("Impossible de calculer la
moyenne.");
        }
        return sum / grades.length;
    }
    public static void main(String[] args) {
        try {
            Student student = new Student();
            student.setGrade(0, 85.5);
            student.setGrade(1, 110); // Cette ligne déclenche une exception
            System.out.println("Moyenne : " + student.calcAvg());
        } catch (IllegalArgumentException e) {
            System.out.println("Erreur : " + e.getMessage());
        } catch (ArithmeticException e) {
            System.out.println("Erreur de calcul : " + e.getMessage());
        }
    }
}
```

Phase de conception
? Élaborer les différentes parties du système et leurs interactions
? Conception architecturale : partitionner le système en sous-systèmes / modules / composants ? Conception détaillée : déf contenu des modules
identifiés (classes, collaborations, comportements, etc.)

Phase de conception ? Objectifs de la conception orientée objet
On parle d'un processus de découverte itératif
? Identifier les classes ? Identifier les responsabilités de ces classes ? Identifier les relations entre ces classes

Phase de conception
7
? Documenter la conception
? Un plan qui facilite l'implémentation et la maintenance du logiciel ? Plusieurs artéfacts de conception
? Diagrammes de classes ? Diagrammes de collaborations ? Diagrammes d'états ? Description textuelle

```
abstract class A {
  public abstract void foo() throws IOException
}

class B extends A {
  @Override
  public void foo() throws SocketException {
    System.out.println(« Foo de Socket");
  } // autorisé

  @Override
  public void foo() throws SQLException {
    System.out.println(« Foo de SQL");
  } // non autorisé
}
```

Une méthode redéfinie ne peut pas lever plus d'exceptions que la méthode de la super-classe.