Depuis J5 une méthode peut avoir un nombre variable d’arguments : Type …  
public static void affiche (String ... list){//list peut être manipulé comme un tableau}

## Règles de visibilité des noms

Un nom introduit dans un bloc de classe peut être redéfini dans un bloc imbriqué dans celui-là.  
=> Dans ce cas, dans le bloc imbriqué, le nom est caché par sa redéfinition

Un nom introduit dans un bloc qui n’est pas de classe est visible dans le bloc, y compris ses blocs imbriqués, mais uniquement à partir de son introduction. Il n’est pas visible en dehors du bloc. Un nom introduit dans un bloc qui n’est pas de classe ne peut pas être redéfini plus loin dans le même bloc (même pas dans un de ses blocs imbriqués)

## Modificateurs

### modificateur d’acces de classe :

■ public : classe visible à toute autre classe  
 ■ pas de modificateur : visibilité package

■ abstract : incomplète, pas d’instance  
 ■ final : pas d’extension

Tous les champs (même privés) sont accessibles à l’intérieur de la classe

### Contrôle d’accès de membres (constructeurs+blocs d'initialisation ne sont pas des membres) :

■ private (visible uniquement par la classe) ■ public (visible par tout le monde)   
 ■ pas de modificateur (package) (visible uniquement par les classes du même package)  
 ■ protected (visible uniquement par les classes du même package et par les sous-classes)

■ static (membre commun à la classe)  
 ■ final (champ constant / classe non extensible / méthode non redefinissable)

une sous-classe :  
■ n’a pas accès aux membres private de la super-classe  
■ a accès aux membres avec visibilité package uniquement si elle se trouve dans le même package que la super-classe  
■ a accès aux membres protected et public de la super-classe

## Constructeurs

this() pour appeler un constructeur dans un autre constructeur, si présent doit être la première instruction

class Point {  
 public Point(double pX, double pY){}  
 public Point () {this (0,0);}  
 public Point (Point p) {this (p.x, p.y);  
}

## Constructeurs de sous-classes

Initialisation d’un objet de sous-classe : à la fois les champs hérités et les champs propres doivent être initialisés

Pour initialiser les champs hérités il est souvent nécessaire d'invoquer le constructeur de la classe parent  
Si présent, super() DOIT ê la première instruction  
■ super() et this() ne peuvent pas ê invoqués dans le même constructeur  
■ si aucun n’est invoqué, appel implicite à super() sans arguments  
■ si il n’existe pas, ERREUR

## Héritage

fille hérite toutes les propriétés et méthodes de mère – réutilisation : elle représente et manipule uniquement la “différence”

Polymorphisme : un Employé est une Personne => un objet Employé  
=> peut être utilisé partout où on s’attend une Personne (mais pas vice-versa=)

## Redéfinition, overriding, overloading

■ Un champ avec le même nom d’un champ de la super-classe cache ce dernier  
=> il faut utiliser super.variable pour y avoir accès  
=> pareil pour les méthodes et l'overriding

■ Le type de retour d’une fonction n’est pas partie de la signature  
=> peut changer dans une redéfinition, restriction : doit être un sous-type de l’original

■ Le modificateur d’accès peut changer dans une redéfinition  
=> restriction : uniquement pour élargir l’accès

■ Le modificateur final interdit la redéfinition pour une méthode  
=> pas pour les variables

## Type déclaré et type effectif

Truc objet = new Machin();

### ■ Type déclaré :

associé au nom de la variable en phase de compilation, ne change jamais  
=> type déclaré de objet → Truc

### ■ Type effectif :

classe avec laquelle l’objet référencé a été construit, peut changer dynamiquement  
=> type effectif de objet → Machin  
Un objet de type effectif C est considéré également de tous les types de plus haut niveau dans la hiérarchie d’heritage  
=> objet instanceof A == true si A super [type effectif]

## Dynamic vs static binding

■ dynamic binding : le type effectif détermine la version à utiliser  
■ static binding : le type déclaré détermine la version à utiliser

le compilateur ne connait que le type déclaré des variables,  
=> le type déclaré dit ce qu’on peut faire avec l’objet

■ méthodes : dynamic  
■ méthodes statics : static  
■ variables : static  
■ les méthodes en dynamic n'ont pas accès aux variables du type déclaré…

## Classes membres

### non-static -> objet/instance englobant

On DOIT passer par un objet de la classe membre  
=> CEngl.CMembre var = new objCEngl.Cmembre()  
=> il en garde une ref implicite NomClasseEnglobante.this (peut ê occulté)  
=> accès même aux trucs private

### static -> pas associé à une instance

=> accès uniquement aux champs static de Cenglobante  
=> pas de CEnglobante.this

## Classes locales

Classes définies à l'intérieur d'un bloc de code (ex: une méthode)

analogue à des variables locales :  
=> pas membre de la possible classe englobante   
=> visible uniquement dans le bloc de code dans laquelle elle est définie

Une classe locale a accès :  
■ aux membres de la classe englobante  
■ aux variables locales visibles dans le bloc de code englobant, SSI elles soient “effectively final” : jamais modifiées après création (même pas par la classe locale)  
=> raison : une copie de la variable locale est passée implicitement par le compilateur au constructeur de la classe locale, qui aura un champ implicite up initialisé à cette valeur, pour pouvoir y accéder même après sa destruction à la sortie du bloc

## Classes locales anonymes

objet d’une classe anonyme qui hérite d’une classe/interface TypeDeBase

new TypeDeBase (paramètres du constructeur de base) {  
 champs et méthodes additionnelles de la classe anonyme  
};

## Interfaces

collection de signatures de méthodes, définit un type  
■ peuvent peuvent avoir une implémentation par défault -> "default" et/ou être static, peuvent être private (définies, auxiliaires)  
■ peut contenir des champes constants (public static final implicite) et des classes internes mais pas des variables d’instance

une sous-classe peut extends une seule autre classe, mais implements plusieurs interfaces -> même polymorphisme  
=> si conflit de constantes, Origine.nom  
=> si de méthodes, classe mère l'emporte  
=> si conflit entre interfaces, même si une des méthodes est abstraite, il FAUT une redéfinition (on peut utiliser methode() {Interface.super.methode())}

une interface peut extends plusieurs interfaces

On peut déclarer des variables / paramètres d’un type interface, MAIS on ne peut pas créer des objets de ce type :  
■ Comparable p; //variable  
■ public void selectionSort (Comparable [] v) {…} //parametre  
■ p = new Comparable(); //ERREUR

un objet de classe Classe implements Interface est aussi de type Interface

## Expressions lambda et interfaces fonctionnelles

un peu comme un pointeur sur/passage de fonction  
interface avec une seule méthode abstraite (ex: Comparator<T>, Action Listener, Function<T,R>)  
ancienne version : créer un objet d'une classe anonyme avec une méthode

bloc de code, précédé par la liste de tous les paramètres utilisés dans ce code :  
■ (int k) -> { for (int i = k; i >= 0; i--){System.out.println(i);} }  
■ (String first, String second) -> first.length() - second.length()

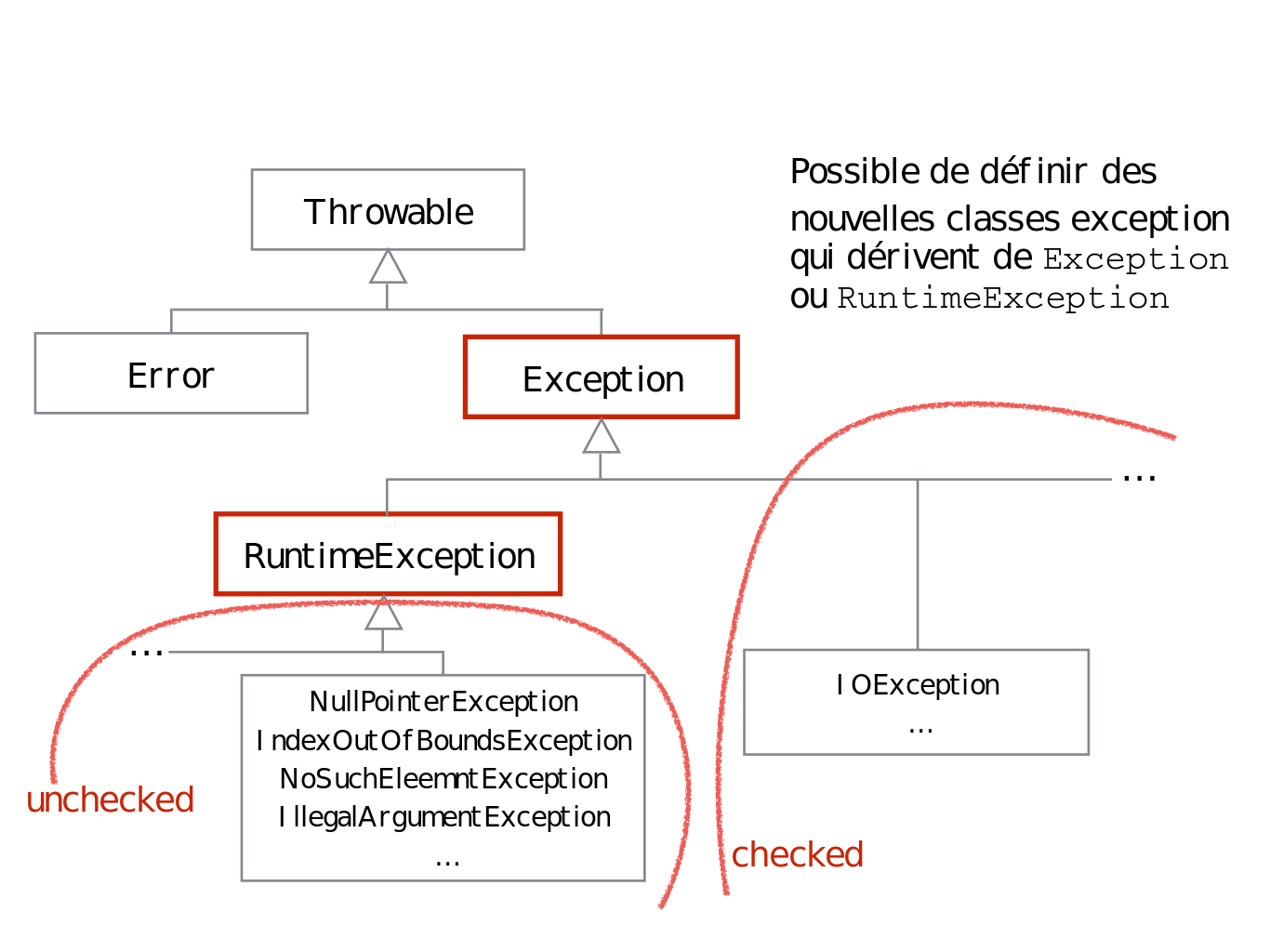
Si le type des paramètres est déterminé par le contexte, il peut être omis

affecter une exp Lambda à une variable de type interface fonctionnelle :  
=> l'exp lambda fournit la définition pour la seule méthode de l'interface

si argument, pas besoin de passer par un objet, on peut envoyer une exp lambda  
compatible comme argument de type InterfaceFonctionnelle  
=> comme la fonction de sort dans python  
=> Interface ActionListener {abstract void actionPerformed(ActionEvent e);}

faire référence à une fonction établie :   
=> nomClasse :: nomMethode OU objet :: nomMethode

visibilité comme une glasse locale : tout de la classe engl + toute variable locale visible "effectively final" dans le bloc englobant

Même règles d’accès aux variables que les classes locales / anonymes !  
=> java crée une instance d’une classe anonyme qui implémente l’interface fonctionnelle -> comme champs implicites des copies des variables locales qu’elle utilise  
=> MAIS différence : this dans une expression lambda fait référence à l’objet de la CEngl, pas à l’objet de la classe anonyme implicitement crée

## Exceptions

Try {…  
} catch (Exception e) {e.printStackTrace();}  
finally {…} //même si il y a un return avant !!

■ Checked : Classes qui dérivent de Exception, mais pas de RuntimeException : dépendent de conditions externes (I/O, fichiers, réseaux,…)  
■ Unchecked : Classes qui dérivent de RuntimeException, erreurs de programmation

Toute méthode qui peut soulever une exception checked sans la catch doit le déclarer : public void fullRead(String filename) throws FileNotFoundException

/!\ bloc catch doivent aller du plus spécifique au plus général (re:types d’exception), sinon compilateur ERREUR

## Généricité

Pour utiliser une classe générique il faut l’ “instancier" = lui fournir un type : la valeur de la variable T  
On appelle l’utilisation d’une classe générique une invocation  
/!\ T extends Object

convention :  
■ E pour les elements d’une collection  
■ K, V pour le clefs et valeurs d’une collection indexée  
■ T, U, S pour les types arbitraires

### Collections

On veut en général garantir l’intégrité des collections (tous les elements du même type)  
Inconvenient des variantes non-génériques : le contrôle de type est à faire “à la main”  
variantes génériques => contrôle de type et casting automatique (par le compilateur)

### Méthode générique

modifiers <T> returnType methode(Type args) {…}

### Bornes

<T extends Classe & Interface> //seulement une classe, en premier

### Covariance

Les types génériques ne sont pas covariants  
=> si A est une sous-type de B, alors C<A> n’est pas un sous-type de C<B> !  
Les tableaux (Array) sont en revanche covariants  
=> si A est une sous-type de B, alors A[] est un sous-type de B[]  
=> contrôle de type implicite à runtime (cast dynamique)

### généricités imbriquées…

class EnsembleOrdonne <E extends Comparable<E>> trop contraignant  
=> class EnsembleOrdonne<E extends Comparable<? super E>> {…}

## Wildcard/type anonymes/<?>

n’importe quel type (extends Object) est compatible avec ‘?’  
=> à utiliser quand aucun autre type dépend du type anonyme

Borne possible, mais par un seul type (i.e. pas de &), et soit extends soit super  
=> simuler la covariance entre collections !

### Capture du wildcard !

Affectation = passage de paramètres ou affectation d’une variable → <? extends A> est considéré par le compilateur comme un type T dont la seule information disponible est : T is/extends ou is/super A  
=> traité de la même manière  
■ super pour écriture  
■ extends pour lecture  
■ C<?> pas d’écriture, lecture uniquement par des variables Object

## Effacement de la généricité

### Raw class

Une déclaration d'un type générique C<E>{...} crée une seule classe C{…} appelée “effacement” (erasure) de la classe générique ou bien classe “brute” (raw class)  
Le type variable est “effacé” et remplacé par Object (ou par A si Suite <E extends A>)  
=> car la machine virtuelle ne connaît pas la généricité  
=> empêche <int> ou <TypePrimitif> car extend pas Objet

Pour garantir une utilisation de la classe brute qui respecte le type effectif, le compilateur introduit des casts.

## Problèmes liés à la généricité

■ Types dynamiques identiques  
C<A> == C<B> pour getClass et instanceof

■ Instanciation impossible  
new E() → (E) new Object() → ERREUR  
=> Workaround : @FunctionalInterface public interface Supplier<T> {T get ();}  
 dans C<E> : methode (Supplier<E> supplier) {E nouveauE = supplier.get();}  
 Ensuite : C<Type> objet; objet.methode(Type::new); …

■ Tableaux impossibles  
<E> … E[] → ERREUR  
=> utiliser ArrayList<E> ou Object[]

■ Tableau d’elements de type générique impossible  
new ArrayList<String>[]; → ERREUR  
=> sauf exception : <?>

■ Référence qu type variable impossible dans un contexte static  
Car après effacement il y a une classe raw, il n’y a qu’une version des champs statiques → pas de ref à ce type dans une méthode ou champ statique de la classe  
=> excéption : les méthodes génériques, qui peuvent en revanche être statiques

■ Overriding  
C’est la signature après effacement qui compte

■ Héritage  
On peut hériter d’une instance d’une classe (ou interface) générique (class D<T> extends C<String>)  
=> Le compilateur implémente ce genre d’héritage de façon à garantir l’overriding des méthodes instanciées de la classe mère (introduction de méthodes “bridge”)

■ Unchecked warnings  
Si utilisation de raw types bc interaction avec vieux code, warnings unchecked  
=> @SuppressWarnings("unchecked")