TA : Types abstraits (de données)

Types

Déf. Un type est un ensemble (possiblement infini) de valeurs et d'opérations sur celles-ci.

Déf. Un type abstrait est un type accessible *uniquement* à travers une interface.

client : le programme qui utilise le TA

implémentation : le programmme qui spécifie le TA

interface : contrat entre client et l'implémentation

En Java
: interface et implémentation souvent dans le même fichier
: interface défini par la signature des méthodes et variables non-privées
: «clients» de droits différents (sous-classe, package) : `interface` n'est pas exactement l'interface de notre définition (ne définit pas le syntaxe des constructeurs)

Nombres naturels

Objets : $N = \{0, 1, \ldots\}$

Opérations :
$\mathsf{add}: N \times N \mapsto N$;
$\mathsf{succ}: N \mapsto N$;
$\mathsf{eq\,?}: N \times N \mapsto \{\text{vrai}, \text{faux}\}$;
$\mathsf{less}: N \times N \mapsto \{\text{vrai}, \text{faux}\}$

Beaucoup d'implantations possibles :
chaînes de chiffres (base 10, `String`), factorisation en primes (`int[]`), $i \in N$ representé par un `char[]` de longueur i

Grande différence dans l'efficacité de l'exécution d'opérations

Nombres naturels — implantation 1

```
public class N {
   private int valeur ;  // implantation basé sur le type int
   public N(String v){this.valeur = Integer.parseInt(v);}
   public N add(N a, N b){
      return new N(a.valeur+b.valeur);
   }
   ...
}
```

implantation facile

problème de représentation (si $v \geq 2^{31}$)

Nombres naturels — implantation 2

```
public class N {
   private String valeur ;  // chaîne de chiffres
   public N(String v){this.valeur = v;}
   public N add(N a, N b){
      // ... l'«algorithme» d'addition sur papier
   }
   ...
}
```

implantation un peu plus compliqué

pas de problème de représentation (au moins jusqu'à $10^{2^{31}}$ — longueur max de `String` est `Integer.MAX_VALUE`)

Nombres naturels — implantation 3

factorisation : $312 = 2^3 \cdot 3 \cdot 13 = 2^3 \cdot 3^1 \cdot 5^0 \cdot 7^0 \cdot 11^0 \cdot 13^1$ représenté par `int[] fact = {3,1,0,0,0,1}`

```
public class N {
   private int[] fact ;  //  factorisation en primes
   public N(String in){
    // factorisation
   }
   public N add(N a, N b){
      // ... algorithme d'addition + factorisation du résultat
   }
   ...
}
```

implantation assez compliqué

par contre, pgcd est facile à calculer :
si $a = \sum_i p_i^{a_i}$ et $b = \sum_i p_i^{b_i}$, alors $\mathsf{pgcd}(a,b) = \sum_i p_i^{\min\{a_i,b_i\}}$.
(avec les nombres premiers $p_1 = 2, p_2 = 3, p_3 = 5, \ldots$)

Collection d'éléments

Files généralisée : collection d'éléments avec deux opérations :

1. insertion d'un nouvel élément

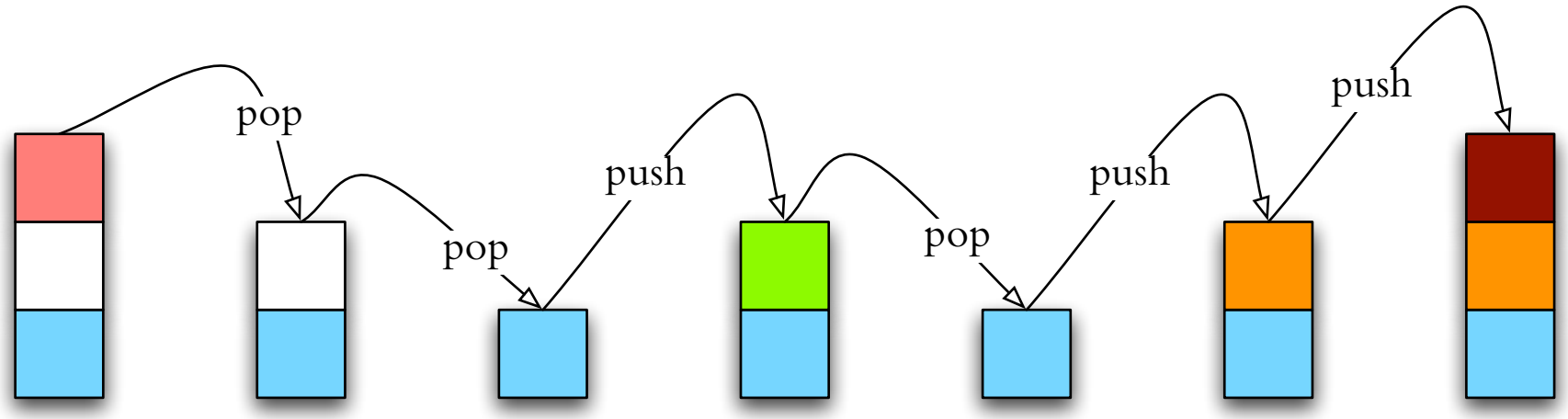
2. suppression d'un élément

il existe beaucoup de variantes selon la politique d'insertion et de suppression d'éléments, ainsi que l'interface pour le TA (comment spécifie-t-on l'«élément» ?)

Pile — idée

Idée de pile : objets empilés l'un sur l'autre, on ne peut accéder qu'à l'élément supérieur

opérations : «empiler» (push) et «dépiler» (pop)

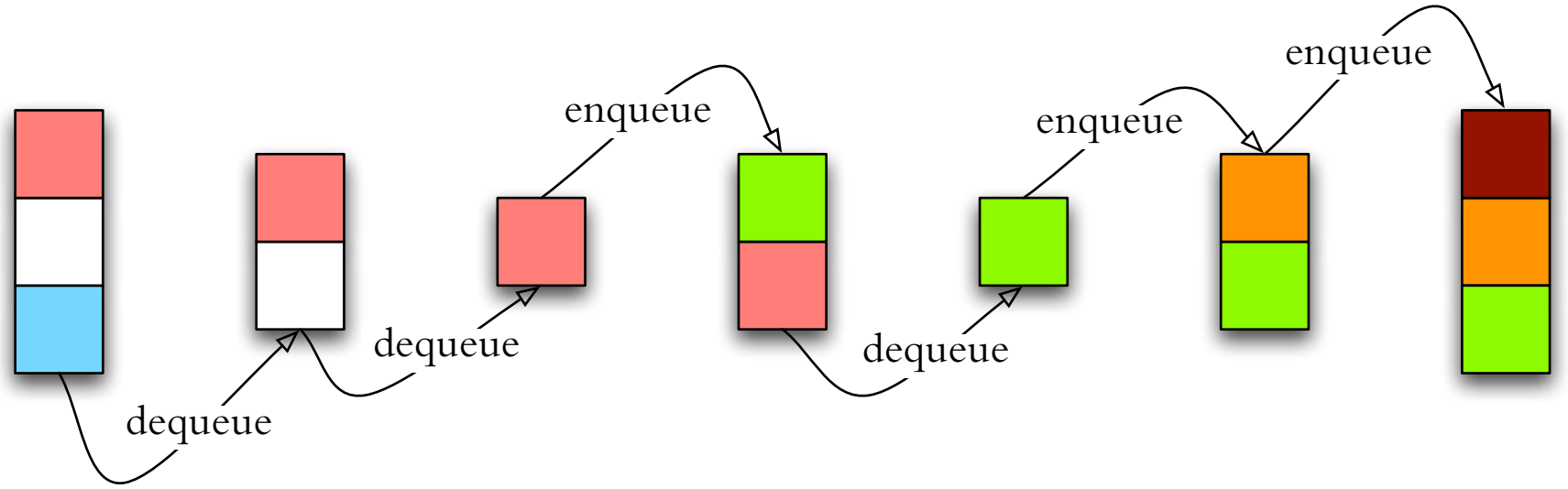


Queue — idée

(queue ou file FIFO, *queue* en anglais)

Idée de queue : comme une file d'attente, objets rangés un après l'autre, on peut enfiler à la fin ou défiler au début

opérations : «enfiler» (enqueue) et «défiler» (dequeue)



TAs classiques

TAs		*opérations principales*						*structures typiques*
		vide?	parcourir	ajouter	retirer	rechercher	min	
sac (*bag*)		+	+	add	-	-	-	tableau ou liste chaînée
files	pile (*stack*)		-	push	pop			
	queue			enqueue	dequeue			
	file à priorités *(priority queue)*		-	add	deleteMin	-	+	tas (*heap*)
ensemble *(set)*			+/-	add ou -	delete ou -	contains	-	tableau de hachage
table de symboles *(symbol table / map)*			+	add	delete	get		
dictionnaire ordonné *(sorted dictionary)*							+	arbre binaire de recherche

Types en Java

Java est un langage fortement typé :

⋆ types primitifs (`int`, `double`, `boolean`, ...)

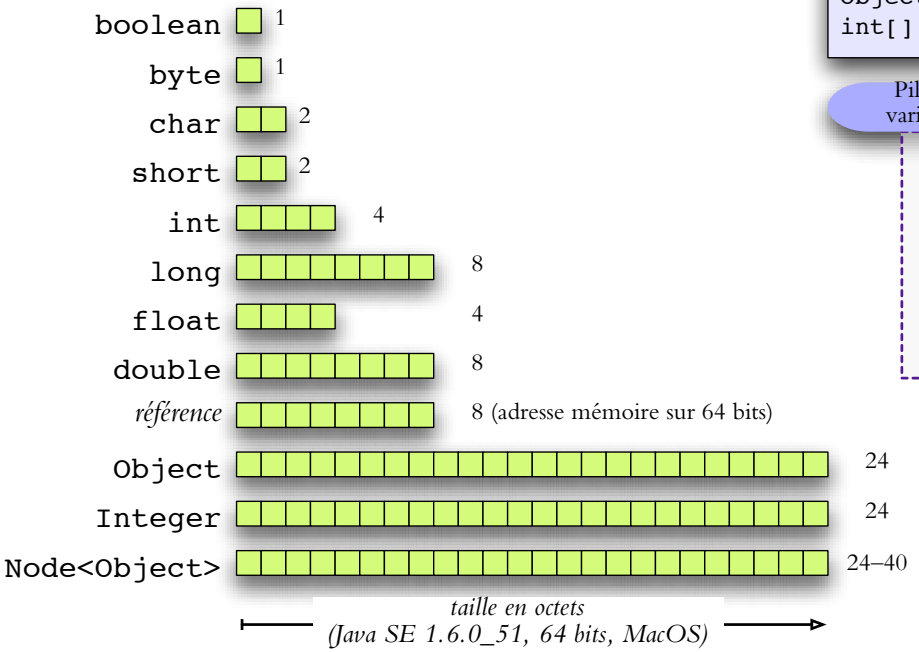
⋆ types agrégés (définis par les classes)

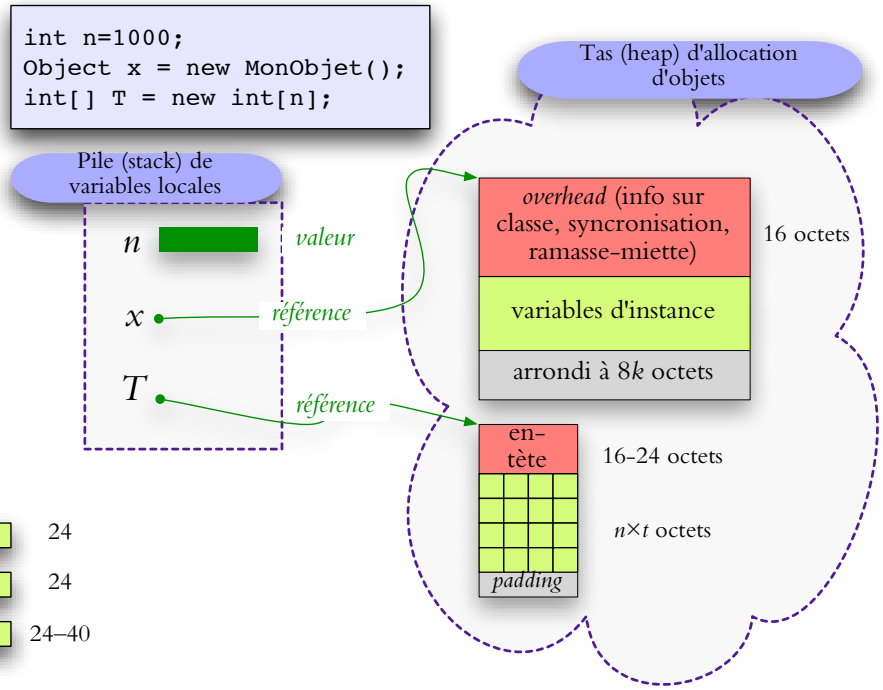
La valeur d'une variable de type agrégé est une référence. Une référence (ou pointeur) est une adresse d'emplacement mémoire contentant de l'information (ou elle est nulle).

En Java, les variables de types simples donnent l'information directement.

Rappel : variable = abstraction d'un emplacement en mémoire (von Neumann)
nom + adresse (lvalue) + valeur (rvalue) + type + portée

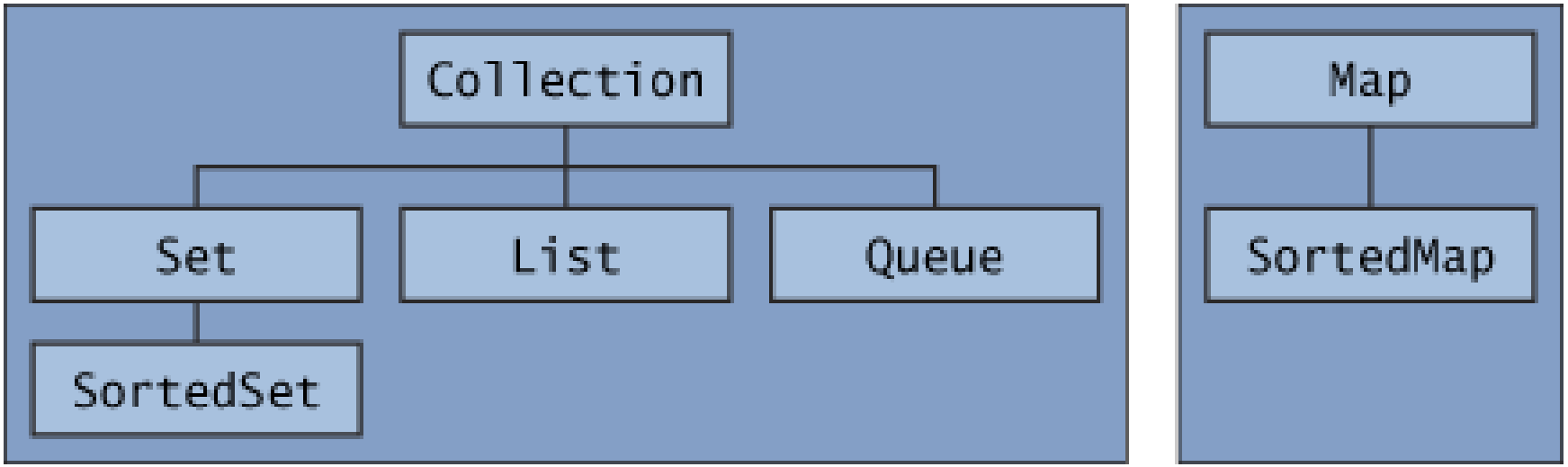
Mémoire en Java





Java Collections

Interfaces fondamentales dans le package `java.util` :



Multiples implémentations

General-purpose Implementations

Interfaces	Implementations				
	Hash table	**Resizable array**	**Tree**	**Linked list**	**Hash table + Linked list**
`Set`	`HashSet`		`TreeSet`		`LinkedHashSet`
`List`		`ArrayList`		`LinkedList`	
`Queue`					
`Map`	`HashMap`		`TreeMap`		`LinkedHashMap`

déclarer le type des variables par l'interface et initialiser par l'implantation choisie

```
Map<Long,String> M = new HashMap<>();// tableau de hachage
Set<Number>      S = new TreeSet<>();// arbre binaire de recherche
```

(types paramétrisés !)

Interface : règles

- ⋆ une interface est toujours publique (même si le mot-clé `public` est omis) ; un modificateur `private` ou `protected` cause une erreur de compilation
- ⋆ les méthodes et variables déclarées sont toujours publiques (même si le mot-clé `public` est omis) ; un modificateur `private` ou `protected` cause une erreur de compilation
- ⋆ les méthodes sont toujours abstraites (même si le mot-clé `abstract` est omis) ; la définition d'une méthode cause une erreur de compilation
- ⋆ les variables sont toujours statiques et finales (même si `static` ou `final` sont omis)
- ⋆ il est interdit de déclarer une variable sans initialisation
- ⋆ il est interdit de déclarer une méthode statique
- ⋆ il est interdit de déclarer un constructeur

java.util.Collection

```
public interface Collection<E> extends Iterable<E>
    // contient des éléments de type E
{
    // opérations de base
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          //optionnelle
    boolean remove(Object element); //optionnelle
    Iterator<E> iterator();

    // opérations de masse
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optionnelle
    boolean removeAll(Collection<?> c);        //optionnelle
    boolean retainAll(Collection<?> c);        //optionnelle
    void clear();                              //optionnelle

    // opérations de tableaux
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

java.util.AbstractCollection

génie logiciel économique : afin d'implémenter `Collection`, il suffit de définir une sous-classe de `AbstractCollection`

— il ne reste que deux méthodes à coder

```
public abstract class AbstractCollection<E>
                    implements Collection<E>
{
    ...
    public abstract Iterator<E> iterator();
    public abstract int size();
    public boolean isEmpty(){ return size()==0;}
    ...
}
```