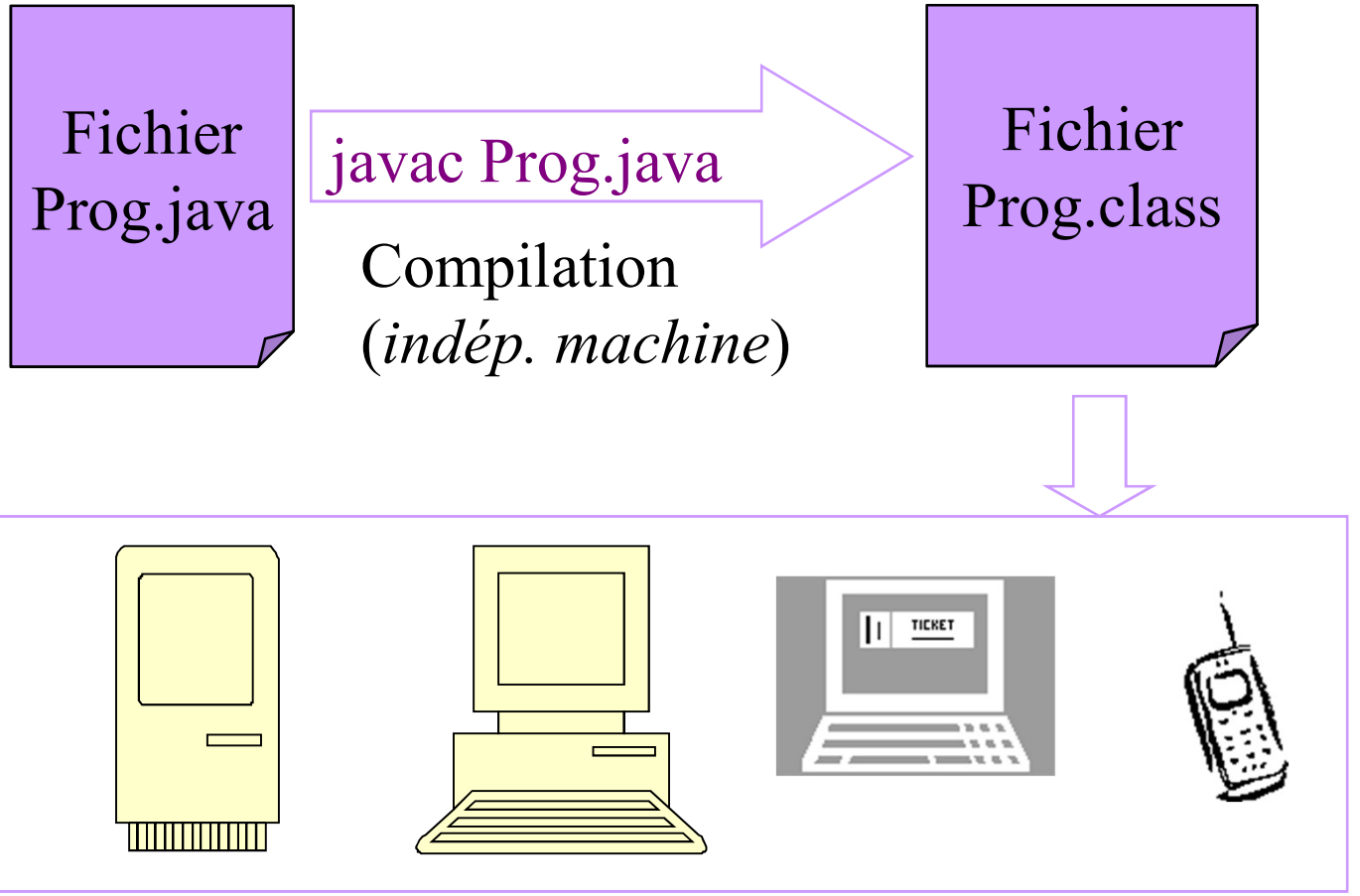
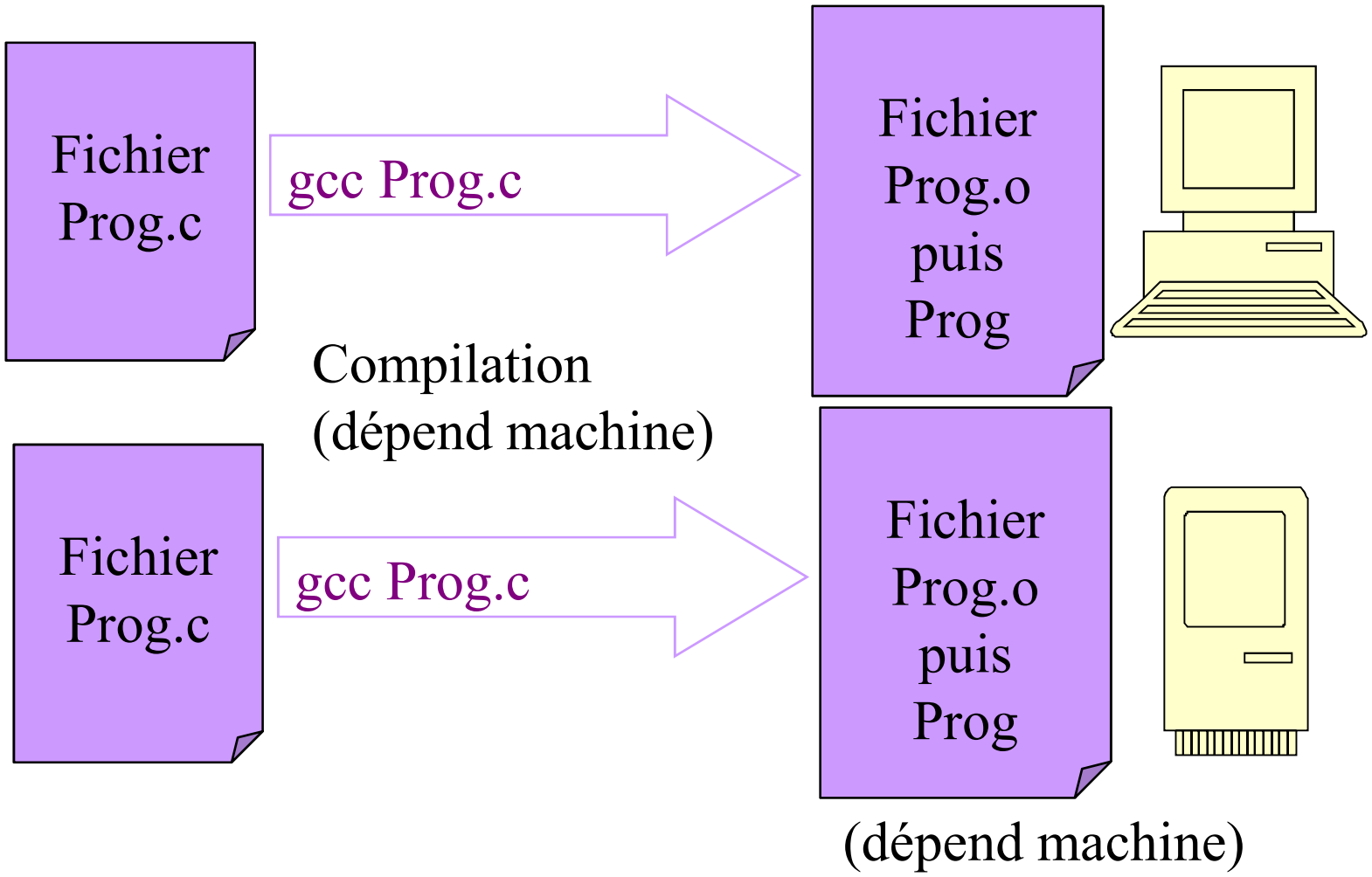
II. Introduction au langage Java

II.3 Compilation de Java

Java : un langage « portable »



Comparaison avec C, C++…



III. Conception de classes en Java

III.1 Abstraction, modularité. Construction de classes simples

Abstraction, modularité

- Une application = plusieurs objets issus de plusieurs classes
  - Comment concevoir ces classes?
- Concepts fondamentaux
  - Abstraction
    - Ignorer les détails pour se concentrer sur un niveau d'analyse supérieur
  - Modularité
    - Division d'un tout en parties
      - bien définies
      - pouvant être étudiées séparément
      - interagissant entre elles de manière bien définie

Exemple : segments et points

- On suppose disposer d'une classe `Point`
  - On ne connaît pas sa réalisation (« implémentation »)
  - On connaît son interface (méthodes publiques)
- On veut réaliser une classe `Segment`
  - On utilisera la classe `Point` ...
  - ... en faisant abstraction de son implémentation.

Point
+Point(float x, float y) +float getX() +float getY() +setX(float x) +setY(float y) +translation(float dx, float dy)

```
class Segment{
        private Point orig,fin;

        public Segment (Point o, Point f){
                this.orig = o;
                this.fin = f;
        }

        public Point getOrigine( ){
                return this.orig;
        }
        public void setOrigine(Point o){
                this.orig = o;
        }
        public void setFin(Point f){
                this.fin = f;
        }
        public void translation(float dx, float dy){
                this.orig.translation(dx, dy);
                this.fin.translation(dx, dy);
        }
        public String toString(){
                return("origine: "+this.orig+" fin: "+this.fin);
        }
}
```

A aucun moment la manière de réaliser la classe `Point` n'intervient dans la construction de la classe `Segment`

Une réalisation de la classe `Point`

```
class Point{
        private float x,y;

        public Point(float abs, float ord){
                this.x=abs;
                this.y=ord;
        }
        public float getX( ){
                return this.x;
        }
        public float getY( ){
                return this.y;
        }
        public void setX(float abs){
                this.x=abs;
        }
        public void setY(float ord){
                this.y=ord;
        }
        public void translation(float dx, float dy){
                this.x+=dx; this.y+=dy;
        }
        public String toString(){
                return("abscisse: "+this.x+" ordonnee: "+this.y);
        }
}
```

Nouvelle version de `Point` (avec *surcharge*)

```
class Point{
        private float x,y;

        public Point(float abs, float ord){
                this.x = abs;
                this.y = ord;
        }

        public Point(float abs){
                this.x = abs;
                this.y = 0.0;
        }

        public Point(){
                this.x = 0.0;
                this.y = 0.0;
        }

        // Autres méthodes inchangées

}
```

Autre version de `Point` (équivalente à la précédente)

```
class Point{
        private float x,y;

        public Point(float abs, float ord){
                this.x=abs;
                this.y=ord;
        }

        public Point(float abs){
                this(abs, 0.0);
        }

        public Point(){
                this(0.0);
        }

        // Autres méthodes inchangées

}
```

`this(x,y)` : appel du constructeur à deux paramètres

Méthode `toString()`

```
...

Point p1 = new Point(0.0, 0.0);
Point p2 = new Point(1.0, 2.0);
Point p3 = new Point(3.0, 4.0);

Segment s1 = new Segment (p1, p2);
Segment s2 = new Segment (p2, p3);

s1.translation(0.5, -0.75);
S2.translation(1.5, 2.0);

System.out.println(p1);
System.out.println(s1);
...
```

Equivalent à:

```
System.out.println(p1.toString());
System.out.println(s1.toString());
```

Méthode `toString` appelée implicitement dès qu'un objet est utilisé comme un `String`

Attributs ou méthodes `static`

```
public class PostIt
{
    static int nb;
    public static final int MAX=100;


    public PostIt()
    {
      nb++;
      System.out.println(« Post-it créés: " + nb);
    }

}
```

- `nb` est commun à tous les objets de la classe PostIt
- On peut utiliser un attribut `static` même si on n'a pas créé d'objet
  - `System.out.println("Nombre MAX: " + PostIt.MAX);`

Attributs ou méthodes `static`

- Il s'agit d'attributs et de méthodes qui sont *associés à la classe* et non pas aux objets.
- La méthode main en est un exemple.
- Dans une méthode static on ne peut pas utiliser des attributs de la classe non static.

III. Conception de classes en Java

III.2 Instructions de contrôle, itérations, tableaux

Instruction conditionnelle

```
public class EquationDeg2{

   private float a, b, c; //ax*x+b*x+c=0

    public EquationDeg2(float a, float b, float c){
       this.a = a; this.b = b; this.c = c;

       if(a == 0) {
         System.out.println("Att. ce n'est pas une
   équation du 2nd deg.");
       } else {
         System.out.println(« OK, équation 2nd deg.");
       }
    }
}
```

Types énumérés

```
public enum Jour {
          LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE;
}
public enum Mois{
          JAN, FEV, MAR, AVR, MAI, JUN, JUL, AOU, SEP, OCT, NOV, DEC;
}
public class DateEx {
          private Jour jour;
          private Mois mois;

          public DateEx(){
                    jour = Jour.LUNDI; mois = Mois.JAN;
          }
          public void afficherJour(){
                    if (jour == Jour.LUNDI) {
                              System.out.println(" Lundi ");
                    }
                    else if // ...
                    ...
          }
          public boolean estOuvré(){
                    return (jour.ordinal() <= Jour.VENDREDI.ordinal()) ;
          }
}
```

Instruction de choix (`switch`)

```
public class DateEx{
  private Jour jour;
  private Mois mois;

  public DateEx(){
    jour = Jour.LUNDI; mois = Mois.JAN;
  }
  public void afficherActivité() {
    switch (jour) {
    case LUNDI:
    case MARDI:
    case MERCREDI:
    case JEUDI:
    case VENDREDI:
      System.out.println("Je travaille");
      break;
    case SAMEDI:
      System.out.println("Je fais les courses");
      break;
    case DIMANCHE:
      System.out.println("Je me promène");
      break;
    default: // optionnel
      System.out.println("Jour invalide");
     }
   }
}
```

Itération while et for

```
public class Exemple{

  public Exemple(){}

  public void méthode1(int n) {
        int s = 0;
        int i = 1;
        while (i <= n){
              s+=i;
              i++;
        }
  }
  public void méthode2(int n) {
        int s = 0;
        for (int i = 1; i <= n; i++){
              s+=i;
        }
  }

}
```

Tableaux

- Les tableaux peuvent contenir des éléments de type primitif ou bien des objets

- Syntaxe spéciale

Exemple : une pile d'entiers

```
public class PileEntiers
{
    private int[] pile;
    private int sommet;

    public PileEntiers(int tailleMax)
    {
        pile= new int[tailleMax];
        sommet = 0;
    }
    public boolean pileVide(){return(sommet == 0);}

    public void empiler(int x){pile[sommet++]=x;}

    public int dépiler(){return pile[--sommet];}
}
```

Déclaration de tableau

Création d'objet tableau

Tableaux constants, taille d'un tableau

déclaration et initialisation (seule utilisation possible)

```
…
private int[] numbers = { 3, 15, 4, 5 };
...
for (int i=0; i<numbers.length; i++){
        System.out.println(numbers[i]);
}
```

Donne la taille d'un tableau (ici: 4)

Arguments de la ligne de commande

```
public class ExempleArgs{

    public static void main(String[] args)
    {
     for(int i = 0; i < args.length; i++)
            System.out.println(args[i]);
    }
}
```

Bilan des concepts introduits

- Abstraction et modularité
- Surcharge (polymorphisme)
- toString
- “static”
- Instructions de contrôle
- Tableaux

III. Conception de classes en Java

III.3 Collections

Collections

- Les *collections* sont des objets qui peuvent stocker une quantité arbitraire d'autres objets

- Java offre plusieurs moyens de réaliser des collections
  - Tableaux
  - Listes, tables, arbres, …

- Nous comprendrons mieux le concept de collection quand nous aurons abordé la *notion d'interface* (voir plus loin).

Exemple : réalisation d'un bloc-notes à l'aide d'une collection

- Classe `Notebook` permettant de stocker un nombre quelconque de notes
- Les notes doivent pouvoir être consultées individuellement
- On souhaite connaître le nombre des notes

```
import java.util.ArrayList;

public class Notebook
{
   private ArrayList<String> notes;

    public Notebook()
    {
        notes = new ArrayList<String>();
    }

    ...
}
```

On importe la classe java.util.ArrayList

La classe `ArrayList` : une collection toute prête

- La classe `ArrayList` du paquetage `java.util`
  - constructeur
    `ArrayList()`
      - Construit une liste vide.
  - méthodes
    - `add(Object o)`
      - Ajoute un objet en fin de liste.
    - `get(index i)`
      - Retourne l'élément d'index i (index du premier élément = 0).
    - `size()`
      - Nombre d'éléments de la liste.

  …

Caractéristiques d'`ArrayList`

- Sa capacité augmente autant que nécessaire.
- Elle conserve les objets dans l'ordre
- On n'a pas besoin de savoir comment tout cela est réalisé.
- On doit spécifier:
  - le type de la collection : `ArrayList`
  - le type des objets qu'elle contient : `<String>`
- On dit "ArrayList de String".

Classes “génériques”

- On dit qu’une collection est un type *générique* (ou *paramétré*).
  - Nous allons apprendre, plus loin dans ce cours, comment construire des classes génériques.

- `ArrayList` implémente une liste
  - `add`, `get`, `size`, etc.

- Le paramètre précise le type des objets de la liste:
  - `ArrayList<Person>`
  - `ArrayList<TicketMachine>`
  - …

Utilisation de la collection

```
public class Notebook
{
    private ArrayList<String> notes;
    ...

    public void storeNote(String note)
    {
        notes.add(note);
    }

    public int numberOfNotes()
    {
        return notes.size();
    }

    ...
}
```

Ajout d'une note

Retourne la taille de la collection

Consulter les objets de la collection

```
public void showNote(int noteNumber)
{
    if(noteNumber < 0) {
        // This is not a valid note number.
    }
    else if(noteNumber < numberOfNotes()) {
        System.out.println(notes.get(noteNumber));
    }
    else {
        // This is not a valid note number.
    }
}
```

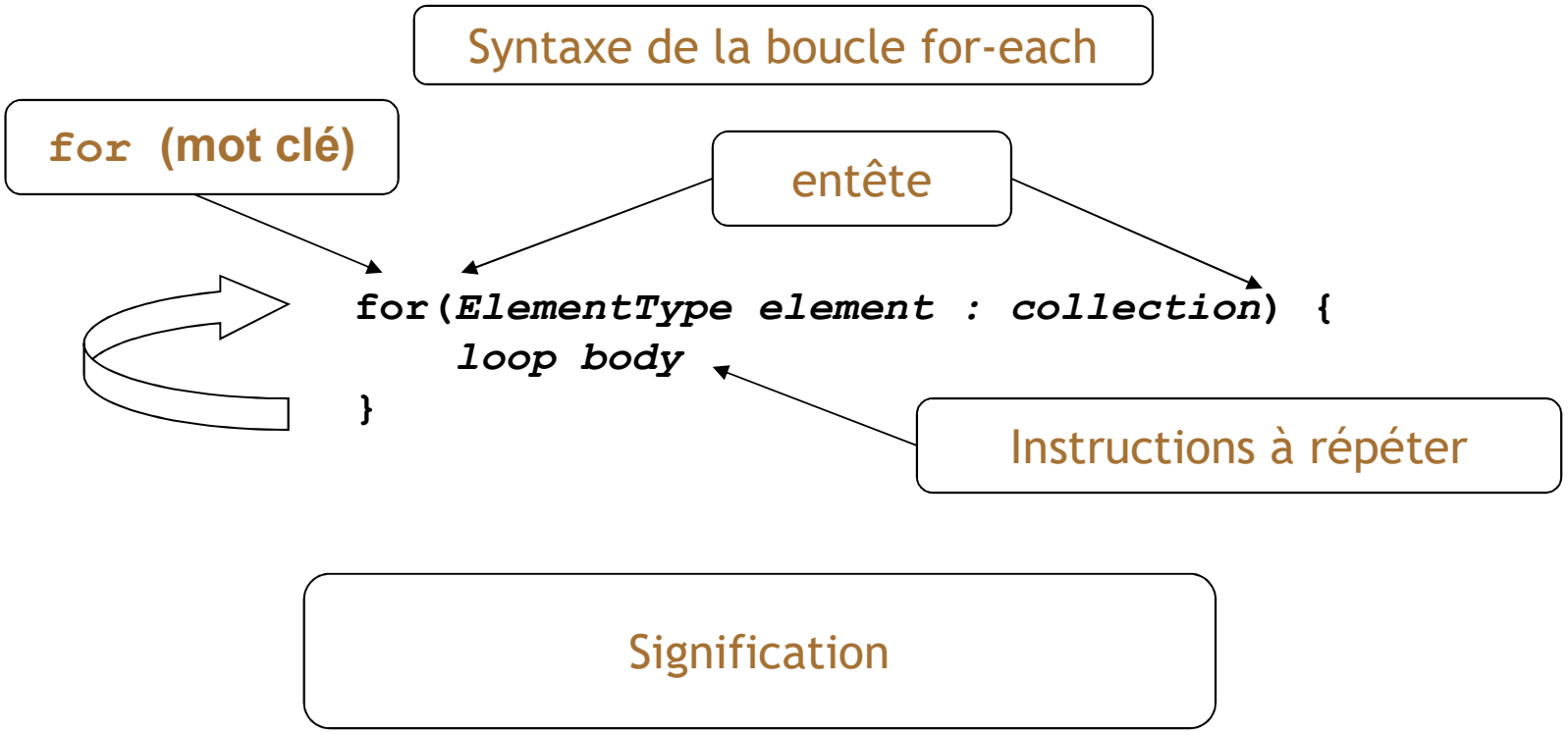
Vérification index

Impression de la note

Itérations et collections

- `ArrayList` n'est pas la seule classe collection :
  - `HashSet`, `TreeSet`…
- Java offre plusieurs possibilités *d'écrire une itération sur une collection*
  - *"for-each"*
    - Possibilité de répéter une action pour chaque objet d'une collection
  - …

For-each



Pour chaque *element* de la *collection*, exécuter *loop body*.

Exemple

```
public class Notebook
{
    private ArrayList<String> notes;
    ...


public void listNotes()
{
    for(String note : notes) {
        System.out.println(note);
    }
}
```

Pour chaque *note* dans *notes*, afficher *note*

Itérations

- Java offre plusieurs possibilités d'écrire une itération
  - *for-each*
    - Possibilité de répéter une action pour chaque objet d'une collection
  - *While, for*

Exemple

```
public class Notebook
{
  private ArrayList<String> notes;
  ...


  public void listNotes()
  {
    int index = 0;
    while(index < notes.size()) {
      System.out.println(notes.get(index));
      index++;
    }
  }
}
```

Recherche dans une collection

```
int index = 0;
boolean found = false;

while(index < notes.size() && !found) {
    String note = notes.get(index);
    if(note.contains(searchString)) {
        found = true;
    }
    else {
        index++;
    }
}
```

Itérations

- Java offre plusieurs possibilités d'écrire une itération
  - *for-each*
    - Possibilité de répéter une action pour chaque objet d'une collection
  - *While*
  - *Itérateurs*

Utilisation d'itérateurs

`java.util.Iterator`

retourne un objet `Iterator`

```
Iterator<ElementType> it = myCollection.iterator();

while(it.hasNext()) {
     call it.next() to get the next object
     do something with that object
}

public void listNotes()
{
     Iterator<String> it = notes.iterator();
     while(it.hasNext()) {
          System.out.println(it.next());
     }
}
```

Bilan des concepts introduits

- Collections
- Classes “génériques”
- ArrayList
- Itération for-each, itérateurs