LP Révision

Maria-Iuliana Dascalu
mariaiuliana.dascalu@gmail.com

Révision sur les objets en java

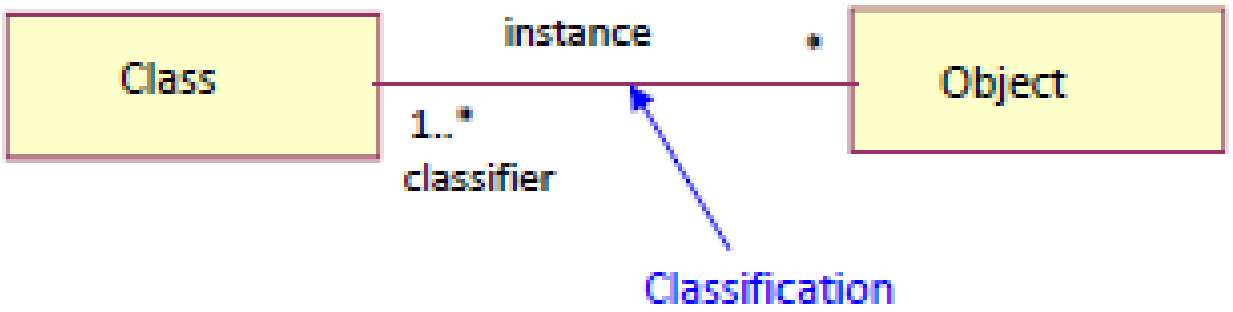
- Objets dans la programmation orientée objet sont des représentations d'instances de concepts sur l'ordinateur.
- La modélisation objet dans nos programmes consiste à créer une représentation abstraite, sous forme d'objets, d'entités ayant une existence matérielle (arbre, personne, téléphone, ...) ou bien virtuelle (sécurité sociale, compte bancaire, ...). .
- Trois caractéristiques essentielles des objets dérivés à partir des caractéristiques des entités réelles sont les suivantes:
  - **Les attributs** (on parle parfois de propriétés): Il s'agit des données caractérisant l'objet. Ce sont des variables stockant des informations d'état de l'objet
  - **Les méthodes** (appelées parfois fonctions membres): Les méthodes d'un objet caractérisent son comportement, c'est-à-dire l'ensemble des actions (appelées opérations) que l'objet est à même de réaliser. Ces opérations permettent de faire réagir l'objet aux sollicitations extérieures (ou d'agir sur les autres objets). De plus, les opérations sont étroitement liées aux attributs, car leurs actions peuvent dépendre des valeurs des attributs, ou bien les modifier
  - **L'identité**: L'objet possède une identité, qui permet de le distinguer des autres objets, indépendamment de son état. On construit généralement cette identité grâce à un identifiant découlant naturellement du problème (par exemple un produit pourra être repéré par un code, une voiture par un numéro de série, ...)

Révision sur les classes en java

- On appelle classe la structure d'un objet, c'est-à-dire la déclaration de l'ensemble des entités qui composeront un objet. Un objet est donc "issu" d'une classe, c'est le produit qui sort d'un moule. En réalité on dit qu'un objet est une **instanciation** d'une classe, c'est la raison pour laquelle on pourra parler indifféremment d'objet ou d'instance (éventuellement d'occurrence).
- Une classe est composée:
  - d'attributs: il s'agit des données, dont les valeurs représentent l'état de l'objet
  - des méthodes : il s'agit des opérations applicables aux objets
- Deux instanciations de classes pourront avoir tous leurs attributs égaux sans pour autant être un seul et même objet (c'est la différence entre *état* et *identité*).
- Exemple: si on définit la classe *voiture*, les objets *Peugeot 406*, *Volkswagen Golf* seront des instanciations de cette classe. Il pourra éventuellement exister plusieurs objets *Peugeot 406*, différenciés par leur numéro de série.

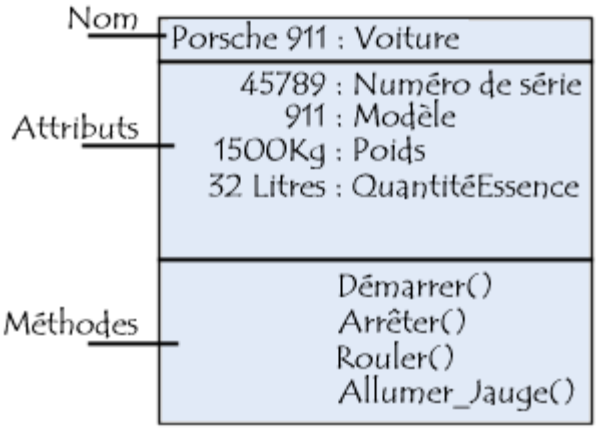
La classification en java

- La classification est l'acte ou le résultat de l'application d'un concept (type) à l'un de ses instances.
- La classification dans la programmation orientée objet peut peut être représentée comme avec des diagrammes UML:



Des diagrammes UML

- UML (*Unified Modeling Language*, que l'on peut traduire par "*langage de modélisation unifié*) est une notation permettant de modéliser un problème de façon standard.
- Dans notre cas, UML est un formalisme de modélisation objet.
- Avec la méthode UML, un objet est par exemple représenté de la façon suivante :

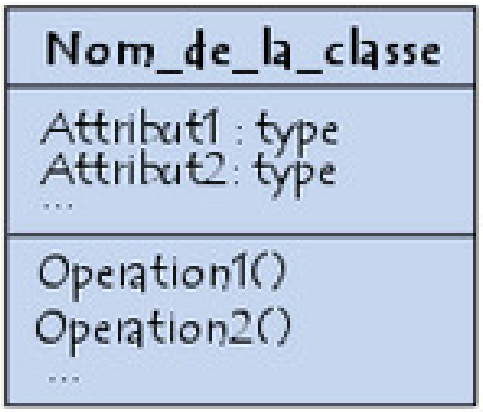


Importance des diagrammes UML

- Un diagramme UML permet de définir le problème à haut niveau sans rentrer dans les spécificités d'un langage, parce que les langages orientés objet constituent chacun une manière spécifique d'implémenter le paradigme objet
- Un diagramme UML représente ainsi un outil permettant de définir un problème de façon graphique, afin par exemple de le présenter à tous les acteurs d'un projet.
- Un diagramme UML est une méthode d'analyse du problème (afin de couvrir toutes les facettes du problème), d'autre part un langage permettant une représentation standard stricte des concepts abstraits (la modélisation) afin de constituer un langage commun.

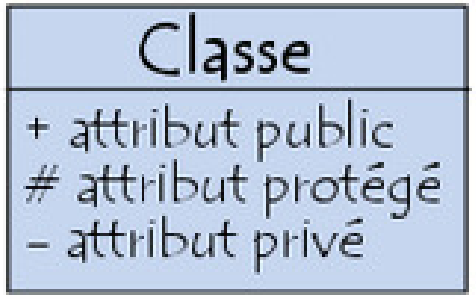
Modélisation des classes en UML

- UML propose une manière de représenter les objets de façon graphique, sous forme de **rectangle, dans lequel le nom de l'objet est souligné:**
  - Le premier contient le nom donné à la classe (non souligné).
  - Les attributs d'une classe sont définis par un nom, un type (éventuellement une valeur par défaut, c'est-à-dire une valeur affectée à la propriété lors de l'instanciation) dans le second compartiment.
  - Les opérations sont répertoriées dans le troisième volet du rectangle.



La visibilité des attributs en UML

- Les niveaux de visibilité des éléments de la classe définissent les droits d'accès aux données selon que l'on y accède par une méthode de la classe elle-même, d'une classe héritière, ou bien d'une classe quelconque. Il existe trois niveaux de visibilité:
  - **publique**: Les fonctions de toutes les classes peuvent accéder aux données ou aux méthodes d'une classe définie avec le niveau de visibilité *public*. Il s'agit du plus bas niveau de protection des données
  - **protégée**: l'accès aux données est réservé aux fonctions des classes héritieres, c'est-à-dire par les fonctions membres de la classe ainsi que des classes dérivées
  - **privée**: l'accès aux données est limité aux méthodes de la classe elle-même. Il s'agit du niveau de protection des données le plus élevé
- La notation UML permet de représenter le niveau de visibilité des attributs de façon graphique en faisant précéder le nom de chaque attribut par un caractère représentant la visibilité:
  - **+** défini un attribut public
  - **#** défini un attribut protégé
  - **-** défini un attribut privé



Relations entre les classes (1)

- **Association** exprime une connexion sémantique bidirectionnelle entre deux classes.
- **Dépendance** est une relation d'utilisation unidirectionnelle et d'obsolescence (une modification de l'élément dont on dépend, peut nécessiter une mise à jour de l'élément dépendant)
- **Spécialisation**
  - Démarche descendante, qui consiste à capturer les particularités d'un ensemble d'objets, non discriminés par les classes déjà identifiées.
  - Consiste à étendre les propriétés d'une classe, sous forme de sous-classes, plus spécifiques (permet l'extension du modèle par réutilisation).
- **Généralisation**
  - Démarche ascendante, qui consiste à capturer les particularités communes d'un ensemble d'objets, issus de classes différentes.
  - Consiste à factoriser les propriétés d'un ensemble de classes, sous forme d'une super-classe, plus abstraite (permet de gagner en généricité).

Relations entre les classes (2)

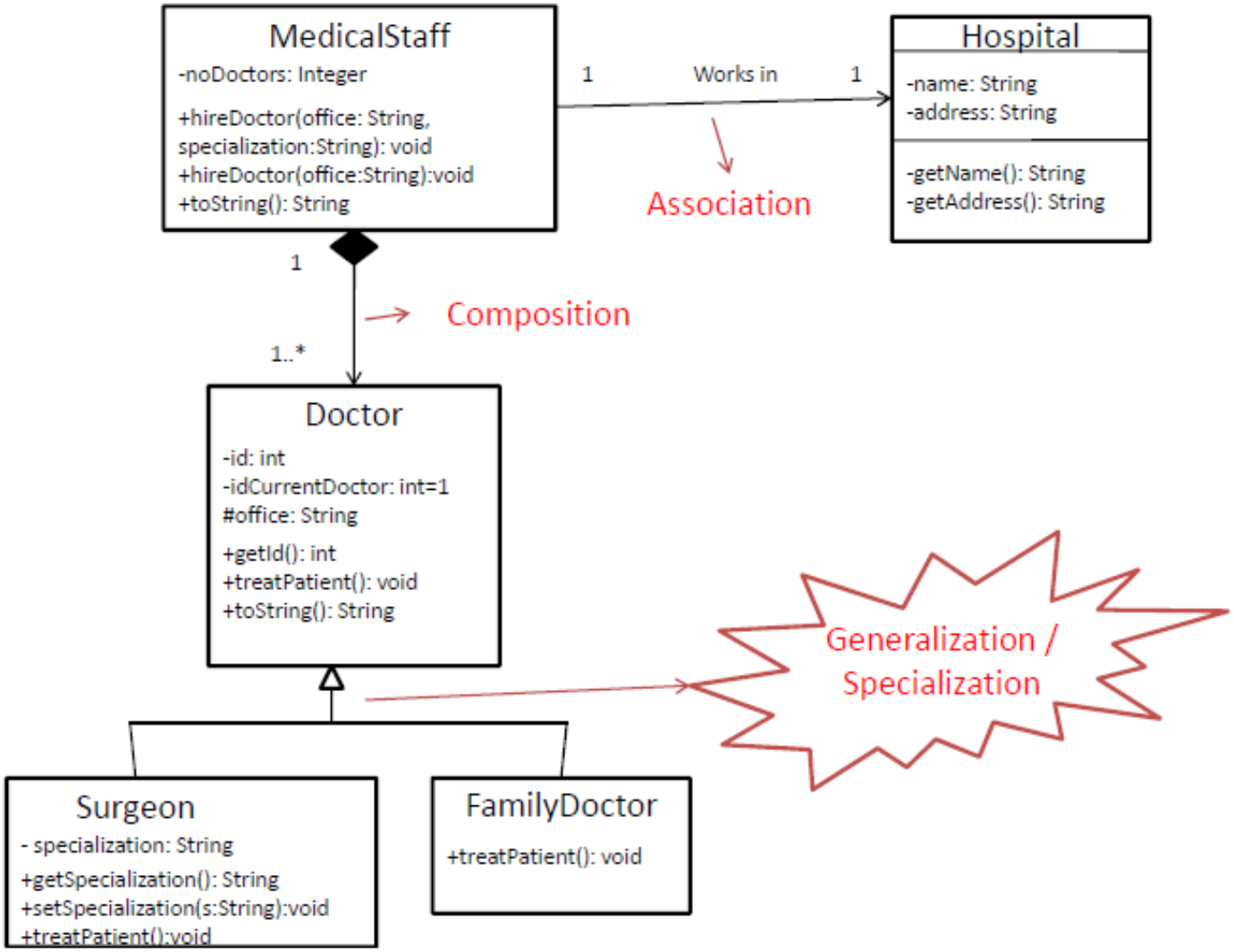
- **L'agrégation** est une association non symétrique, qui exprime un couplage fort et une relation de subordination.
- Elle représente une relation de type "ensemble / élément".
- UML ne définit pas ce qu'est une relation de type "ensemble / élément", mais il permet cependant d'exprimer cette vue subjective de manière explicite.
- Une agrégation peut notamment (mais pas nécessairement) exprimer :
  - qu'une classe (un "élément") fait partie d'une autre ("l'agrégat"),
  - qu'un changement d'état d'une classe, entraîne un changement d'état d'une autre,
  - qu'une action sur une classe, entraîne une action sur une autre.
- A un même moment, une instance d'élément agrégé peut être liée à plusieurs instances d'autres classes (l'élément agrégé peut être partagé).
- Une instance d'élément agrégé peut exister sans agrégat (et inversement): les cycles de vies de l'agrégat et de ses éléments agrégés peuvent être indépendants.

Relations entre les classes (3)

- La composition est une agrégation forte (agrégation par valeur).
- Les cycles de vies des éléments (les "composants") et de l'agrégat sont liés : si l'agrégat est détruit (ou copié), ses composants le sont aussi.
- A un même moment, une instance de composant ne peut être liée qu'à un seul agrégat.
- Les "objets composites" sont des instances de classes composées.

Exemple

Manage Hospital System



Héritage

- Utilisé pour la généralisation / spécialisation relation (« est-un » relation)
- Une sous-classe hérite / étend les champs et les méthodes (qui ne sont pas privés) d'une superclasse:
  - Ajoute de nouveaux champs / méthodes;
  - Modifie (remplacements) des champs / méthodes;
- Mot réservé en Java pour désigner l'héritage: **extends**
- En Java, une classe peut hériter un seul super-classe

Exemple

[modifier] **class** Subclass_Name **extends** Superclass_Name

```
package managehospital;

public class Doctor {
        private int id;
        private static int idCurrentDoctor=1;
        protected String office;

        public Doctor(String office)
        {
                this.office = office;
                this.id=idCurrentDoctor;
                idCurrentDoctor++;
        }

        public int getId() {
                return id;
        }

        public void treatPatient()
        {
                System.out.println("treats patients")
        }

        public String toString()
        {
                return ("The doctor with the id "+this.id+" works in "+this.office+" office.");
        }

}
```

```
public class Surgeon extends Doctor{
        private String specialization;

        public Surgeon(String office)
        {
                super(office);
        }

        public String getSpecialization() {
                return specialization;
        }

        public void setSpecialization(String specialization){
                this.specialization = specialization;
        }

        public void treatPatient()
        {
                System.out.println("operates patients");
        }

}
```

subclass

superclass

adds method

modifies method

Niveaux d'accessibilité

- Des membres publiques sont héritées: la sous-classe «voit» les membres publiques de la super-classe

- Les membres privés ne sont pas héritées: pour accéder aux membres privés de la superclasse dans la sous-classe, nous devons créer getters

- Membres protégées sont héritées: la sous-classe «voit» les membres du public de la superclasse; pour les autres classes, les membres protégées agissent en tant que membres privés

Exemple

```
public class Doctor{
    //subclasses Surgeon and FamilyDoctor can't access
    the id, so we create a getter
    private int id;
    // the office location is "seen" by the two subclasses
    protected String office;
    public int getId(){
          return id;
    }
}
```

Constructeurs

- Le constructeur de la sous-classe appelle toujours le constructeur de la superclasse, explicitement ou implicitement
- Pour éviter les erreurs de compilation, nous avons plusieurs possibilités:
  - Dans le constructeur de la sous-classe, nous devons appeler explicitement l'un des constructeurs de la super-classe (en utilisant le mot-clé **super**)
  - Insérez le constructeur sans paramètres définis dans la super-classe (car il est appelé implicitement par le constructeur sous-classe)

Exemple

```
public Doctor(String office)
{
    //each doctor has to have an office
    this.office = office;
    //each doctor has to have a unique id
    //we use a static field to help us ensuring the uniqueness of each object id
    //within the Doctor class
    this.id=idCurrentDoctor;
    idCurrentDoctor++;
    //it makes NO SENSE to use "this" for a static member
}
public Surgeon(String office)
{
    // calls the explicit constructor of the superclass
    super(office);
}
```

Utilisation d'objets de la sous-classes

```
Surgeon d1=new Surgeon(“AB01”);
// remarquez que la méthode toString n'est pas
// défini dans la sous-classe Surgeon, elle est
// héritée de la super-classe Doctor
System.out.println(d1);
Doctor d2=new Surgeon(“AB06”);
System.out.println(d2);
```

Casting & *instanceof*

Doctor d3=new Surgeon(“AB09”);

d3.setSpecialization(“cardiology”);

$\Rightarrow$Erreur de compilation

- La variable de référence d3 est de type super-classe.
- L'objet auquel il se réfère est de type sous-classe.
- Le compilateur ne sait pas ce truc!
- Solution: *instanceof* utilisation
- Le mot-clé *instanceof* peut être utilisé pour tester si un objet est d'un type spécifié:

  if (objectReference instanceof type)

Polymorphisme

- A caractéristiques d'une entité d'avoir des significations différentes, selon le contexte.
- Exemples de polymorphisme:
  - constructeurs surchargés
  - des méthodes surchargées

  !!! La méthode toString de la classe Doctor est surchargée: elle est définie dans la classe Objectet redéfini dans nos classes.

Surcharge (overloading) vs Substitution (overriding)

- Surcharge:
- nous avons plus des méthodes avec le même nom dans la même classe, mais différents paramètres;
- a lieu lors de la compilation => polymorphisme statique;

- Substitution:
- nous redéfinissons les méthodes de la super-classe au sein de sous-classes, en gardant le même nom, mais en changeant la mise en œuvre
- a lieu lors d'exécution=> polymorphisme dynamique;

Mot réservé **final**

- fixation de la mot-clé **final** à une classe => cette classe ne peut pas être héritée

- fixation de la mot-clé **final** à une méthode => cette méthode ne peut pas être héritée

- fixation de la mot-clé **final** au attribut d'une classe => cet attribut ne peut pas être instancié

Les classes abstraites

- Ils n'ont pas de cas directs, mais ils ont sous-classes avec des exemples concrets.
- Ils doivent avoir au moins une méthode abstraite (une méthode dans laquelle aucun algorithme n'est défini et doit être remplacée dans les sous-classes).
- Example:

```
public abstract class AAA{
        ……….
        public abstract void aaa (); // pas de corps
}
```

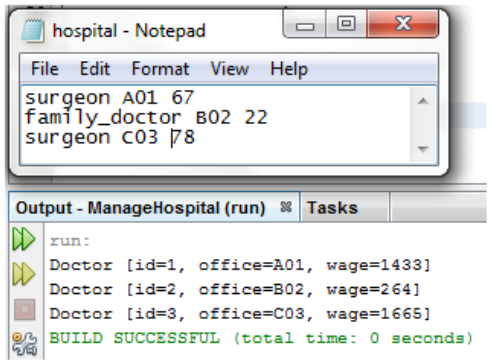
- Vous ne pouvez pas appeler le constructeur d'une classe abstraite!

Exemple

```
public abstract class Doctor {
        //it is seen only inside its class
        //that is why it needs a getter
        private int id;
        // the same for all objects of type Doctor
        private static int idCurrentDoctor=1;
        // it is seen by all subclasses and classes within the package
        protected String office;
        protected long wage;
        protected static long bonus;
        public Doctor(String office, long wage)
        {
                // office an wage are read from the "hospital.txt" file
                this.office = office;
                this.wage = wage;
                // it ensures the uniqueness of an id
                this.id=idCurrentDoctor;
                idCurrentDoctor++;
        }
        public int getId() {
                return id;
        }
        public static void setBonus(long b)
        {
                bonus = b;
        }
       //abstract method
       public abstract void treatPatient();
       // abstract method: has to be implemented in subclasses
       public abstract void calculateWage();
       @Override
       public String toString() {
                return "Doctor [id=" + id + ", office=" + office + ", wage=" + wage
                                + "]";
        }
}
```

```
class Surgeon extends Doctor{
        public Surgeon(String office, long wage)
        {
                // calls the superclass constructor: it is mandatory
                super(office, wage);
        }

        //implements the abstract method from its superclass
        public void treatPatient()
        {
                System.out.println("operates patients");
        }

        //implements the abstract method from its superclass
        public void calculateWage()
        {
                this.wage+=Doctor.bonus+20.100*this.wage;
        }

}
```

```
public class ManageHospital {
	public static void main(String[] args) {
		// instantiate a MedicalStaff object: it has the number of doctors and the hospital
		MedicalStaff mf = new MedicalStaff(3,new Hospital("General Hospital", "Liberty Street"));
		//sets the bonus to be 20
		Doctor.setBonus(20); Doctor[] d = new Doctor[3];
		// read the information from the "hospital.txt" to instantiate the array of doctors
		try{
			//ATTENTION: the txt file has to be within the project folder
			FileInputStream fstream = new FileInputStream("hospital.txt");
			// Get the object of DataInputStream
			DataInputStream in = new DataInputStream(fstream);
			// we memorize in a buffer the file content
			BufferedReader br = new BufferedReader(new InputStreamReader(in));
			// the buffer has more lines and we have to read it line by line
			String strLine;
			int i = 0;
			//Read File Line By Line and parse it
			while (((strLine = br.readLine()) != null)&& (i<3))   {
				// in this case, the separator is blank, but it can be any other character
				String[] words = strLine.split(" ");
				if (words[0].equals("surgeon"))
				{
					d[i++]=new Surgeon(words[1],Long.parseLong(words[2]));
				}
				else
				{
					d[i++]=new FamilyDoctor(words[1],Long.parseLong(words[2]));
				}
			}
			//Close the input stream
			in.close();
			  }
			catch (Exception e1){//Catch exception if any
			  System.err.println("Error: " + e1.getMessage());
			  }

		// for each doctor in the array we calculate the wage
		// then we add it to the array of doctors of the hospital
		for (Doctor i:d)
		{
			i.calculateWage();
			mf.hireDoctor(i);
		}

		mf.display();
	}
}
```



```
class MedicalStaff {
        private int noDoctors;
        private Doctor[] doctors;
        public final Hospital hospital;
        public MedicalStaff(int noDoctors, Hospital hospital)
        {
                //a medical staff works in a hospital (we can't change the name of the hospital, after initialization
                this.hospital=hospital;
                // a medical staff has a collection of doctors
                // at the beginning, the collection is empty
                // we have to fill it = > create a method hireDoctor
                this.doctors = new Doctor[noDoctors];           }
        public Doctor[] getDoctors() {
                return doctors;}
        public void hireDoctor(Doctor d)
        {
                // when we instantiate an object of type MedicalStaff, the noDoctors is 0
                // then, each time a doctor is added, the noDoctors++
                doctors[noDoctors++]=d;}
        public void display()
        {
                // display all the doctors from the medical staff
                for (Doctor i:doctors)
                {
                        // the class doctor has an overriden toString
                        System.out.println(i);}         }
}
```

```
class Hospital {
        private String name;
        private String address;

        public Hospital (String name, String address){
                this.name=name;
                this.address=address;
        }
        public String getName(){
                return this.name;
        }

        public String getAddress(){
                return this.address;
        }
}
```

```java
public abstract class Doctor {
	//it is seen only inside its class
	//that is why it needs a getter
	private int id;
	// the same for all objects of type Doctor
	private static int idCurrentDoctor=1;
	// it is seen by all subclasses and classes within the package
	protected String office;
	protected long wage;
	protected static long bonus;

	public Doctor(String office, long wage)
	{
		// office an wage are read from the "hospital.txt" file
		this.office = office;
		this.wage = wage;
		// it ensures the uniqueness of an id
		this.id=idCurrentDoctor;
		idCurrentDoctor++;
	}

	public int getId() {
		return id;
	}

	public static void setBonus(long b)
	{
		bonus = b;
	}

	//abstract method
	public abstract void treatPatient();

	// abstract method: has to be implemented in subclasses
	public abstract void calculateWage();

	@Override
	public String toString() {
		return "Doctor [id=" + id + ", office=" + office + ", wage=" + wage + "]";
	}
}
```

```java
class Surgeon extends Doctor{
	private String specialization;
	public String getSpecialization() {
	return specialization;
	}
	public void setSpecialization(String specialization) {
	    this.specialization = specialization;
	}
	public Surgeon(String office, long wage)
	{
		// calls the superclass constructor: it is mandatory
		super(office, wage);
	}
	//implements the abstract method from its superclass
	public void treatPatient()
	{
		System.out.println("operates patients");
	}
	//implements the abstract method from its superclass
	public void calculateWage()
	{
		this.wage+=Doctor.bonus+20.100*this.wage;
	}
}
```

```java
class FamilyDoctor extends Doctor{

	public FamilyDoctor(String office, long bonus)
	{
		// calls the superclass constructor: it is mandat
		super(office, bonus);
	}
	//mandatory: implements the abstract method from its supe
	public void treatPatient()
	{
		System.out.println("gives prescriptions");
	}
	//implements the abstract method from its superclass
	public void calculateWage()
	{
		this.wage+=Doctor.bonus+10.100*this.wage;
	}
}
```