# Instrucciones de presentación

Crea una carpeta llamada jPerez*-p01,* donde ***jPerez*** es tu identificador (por ejemplo, si tu nombre es juan perez tu archivo será **jPerez-p0**1) y copia todos tus archivos de código fuente .java en esta carpeta. No copies los archivos .*class* ni ningún otro archivo. A continuación, comprime la carpeta jPerez*-p01* creando un **archivo .zip** llamado jPerez*-p01* (por ejemplo, jPerez*-p01* p).

# Objetivos de aprendizaje

1. Leer los diagramas de clases UML y convertir el diagrama en clases Java .
2. Identificar e implementar relaciones de dependencia, agregación, herencia y composición. .
3. Utilizar correctamente los modificadores de accesibilidad public, private y protected .
4. Escribir código Java para anular métodos .
5. Reconocer cuándo hay herencia entre clases en un OOD. .
6. Diseñar e implementar clases utilizando la herencia. .
7. Escribir código Java para implementar el polimorfismo en una jerarquía de herencia de clases.
8. Implementar una interfaz Java .

# Antecedentes

East Coast State University1 está situada en la hermosa Costa Oeste de California. En ECSU hay dos (2) categorías de estudiantes: estudiantes en el campus y estudiantes online. Los(as) estudiantes del campus se clasifican como residentes (R) o no residentes (N) dependiendo de si residen en el estado o fuera de él. La matrícula básica para los(as) estudiantes del campus es de $7.575 para los residentes y de $14.875 para los no residentes. A algunos(as) estudiantes internos(as), matriculados(as) en determinados programas preprofesionales, como Derecho, Odontología o Farmacia, se les cobra una tasa adicional que varía en función del programa. Los(as) estudiantes universitarios(as) pueden matricularse de hasta 18 horas de crédito a la tarifa básica, pero por cada hora de crédito que supere las 18, pagan una tasa adicional de $475 por cada hora de crédito que supere las 18. Los(as) estudiantes online no son ni residentes ni no residentes. Más bien, su matrícula se calcula como el número de horas de crédito en las que están matriculados multiplicado por la tasa por hora de crédito online, que es de $950 por hora de crédito. Además, algunos(as) estudiantes online matriculados(as) en determinados programas de grado pagan una tasa de tecnología online de $75 por semestre.

# Requisitos del software

Los requisitos del software para este proyecto son

1. La información de los estudiantes de East Coast State se almacena en un archivo de texto llamado *p02-students.txt*. Hay un registro de estudiante por línea, donde el formato de un registro de estudiante para un estudiante en el campus es:

### C id last-name first-name residency program-fee credits

Donde:

C Identifica al(la) estudiante como estudiante del campus.

***id*** El número de identificación del(la) estudiante. Una cadena de 13 dígitos.

***last-name*** El apellido del(la) estudiante. Una cadena contigua de caracteres.

***first-name*** El nombre del(la) estudiante. Una cadena contigua de caracteres.

***residency*** R si el estudiante es residente, N si es no residente.

***program-fee*** Una cuota de programa, que puede ser cero.

***credits*** El número de horas de crédito en las que está matriculado el estudiante.

1 Ver [Blobfish](https://www.smithsonianmag.com/science-nature/behold-the-blobfish-180956967/)!

El formato del expediente de un(a) estudiante online es el siguiente:

### O id last-name first-name tech-fee credits

donde O identifica al estudiante como un estudiante en línea, y el ***id***, ***last-name***, ***first-name***, y ***credits*** son los mismos que para un estudiante en el campus. El campo ***tech-fee*** es T si el estudiante debe pagar la cuota tecnológica o - si al(la) estudiante no se le aplica la tasa tecnológica. He aquí un ejemplo *p02-students.txt* file:

donde O identifica al(la) estudiante como estudiante online, e ***id***, ***last-name***, ***first-name***, y ***credits*** son los mismos que para un(a) estudiante presencial. El campo cuota ***tech-fee*** si el(la) estudiante debe pagar la cuota tecnológica o - si el(la) estudiante no debe pagar la cuota tecnológica. Aquí tienes un ejemplo de archivo *p02-estudiantes.txt*

**Muestra *p02-students.txt***

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| C | 8230123345450 Simons | Jenny | R | 0 | 12 |
| C | 3873472785863 Cartman | Eric | N | 750 | 18 |
| C | 4834324308675 McCormick | Kenny | R | 0 | 20 |
| O | 1384349045225 Broflovski | Kyle | - | 6 |  |
| O | 5627238253456 Marsh | Stan | T | 3 |  |

1. El programa leerá el contenido de *p02-estudiantes.txt* y calculará la matrícula de cada estudiante.
2. El programa escribirá los resultados de la matrícula en un archivo de salida llamado *p02-tuition.txt* con el siguiente formato:

### id last-name first-name tuition id last-name first-name tuition

***...***

donde *id* es el número de identificación del estudiante, *last-name(apellido)* y *first-name(nombre)*, son el nombre del(la) estudiante, y *tuition* (*matrícula)* es la matrícula estimada del(la) estudiante. *id* se mostrará justificado a la izquierda en un campo de tamaño 16, *last-name (apellido)* se mostrará justificado a la izquierda en un campo de tamaño 20, *first-name(nombre)* se mostrará justificado a la izquierda en un campo de tamaño 15, y *tuition* (*matrícula)* se mostrará justificada a la derecha en un campo de tamaño 8 con 2 dígitos después del punto decimal. Para el archivo de entrada de ejemplo, este es el archivo de salida:

**Muestra *p02-tuition.txt***

|  |  |  |  |
| --- | --- | --- | --- |
| 1384349045225 | Broflovski | Kyle | 5700.00 |
| 3873472785863 | Cartman | Eric | 15625.00 |
| 4834324308675 | McCormick | Kenny | 8525.00 |
| 5627238253456 | Marsh | Stan | 2925.00 |
| 8230123345450 | Simons | Jenny | 7575.00 |

1. Los registros del archivo de salida se ordenarán en orden ascendente por *id*.
2. Si el archivo de entrada *p02-estudiantes.txt* no puede abrirse para su lectura (probablemente porque no existe), entonces muestra un mensaje de error en la ventana de salida, cierra los archivos abiertos y termina el programa, por ejemplo,

Sorry, could not open 'p02-students.txt' for reading. Stopping.

(Lo sentimos, no se ha podido abrir 'p02-estudiantes.txt' para su lectura. Se ha detenido.)

1. Si el archivo de salida *p02-tuition.txt* no se puede abrir para escribir, entonces muestra un mensaje de error en la ventana de salida, cierra cualquier archivo abierto y, a continuación, finaliza el programa, por ejemplo,

Sorry, could not open 'p02-tuition.txt' for writing. Stopping.

(Lo sentimos, no se pudo abrir 'p02-tuition.txt' para escribir. Se ha detenido.)

# Diseño de software

Consulta y estudia el diagrama de clases UML de la actividad. Tu programa deberá implementar este diseño.

## Clase principal (Main)

La clase principal se llama *Main* y en el archivo zip del proyecto se incluye una plantilla para *Main.java*. La clase Main contendrá el método *main*() que simplemente instanciará un objeto de la clase *Main* y llamará a *run*() sobre ese objeto. Compila el código de *Main.java* leyendo el diagrama de clases UML, los comentarios e implementando el pseudocódigo.

## Clase TuitionConstants

En el archivo zip del proyecto se incluye una clase llamada *TuitionConstants*. Esta clase declara varias constantes estáticas públicas que se utilizan en otras clases. Las constantes se derivan de lo expuesto en Antecedentes.

## Clase Clasificador (sorter)

Hablaremos de los algoritmos de ordenación más adelante en el curso, así que puede que este código no tenga mucho sentido en este momento. Se ha proporcionado todo el código de ordenación. Si gustas, utiliza el algoritmo de ordenación por inserción, que no es muy eficiente para listas grandes, pero sí para listas pequeñas como la nuestra.

La clase *Clasificador (Sorter)* contiene un método de clase pública *insertionSort*() al que se puede llamar para ordenar una lista de *ArrayList<Student>*. Al ordenar *Student(Estudiantes)* necesitamos poder comparar un *Student A* con otro *Student B* para determinar si *A* es menor o mayor que *B*. Como estamos ordenando por id de *Student*, hacemos que la clase abstracta *Student* implemente la interfaz *java.lang. Comparable*<*Student*> y definimos que el *Student A* es menor que el *Student B* si el campo *mId* de *A* es menor que el campo *mId* de *B*. Así es como ordenamos la lista *ArrayList*<*Student*> por el identificador del estudiante.

*java.lang.Comparable*<T> es una interfaz genérica de Java Class Library (requiere que se especifique un parámetro de tipo *T* cuando se implementa la interfaz) que declara un método:

int compareTo(T obj)

donde *T* representa un tipo de clase y *obj* es un objeto de la clase *T*. El método devuelve un entero negativo si este *T* (el objeto sobre el que se invoca el método) es menor que *obj*, cero si *este T* y *obj* son iguales, o un entero positivo si *este T* es mayor que *obj*. Para hacer que la clase abstracta *Student* implemente la interfaz *Comparable*, escribimos:

public abstract class Student implements Comparable<Student> { ... }

Como Student implementa *Comparable<Student>*, siempre que se llame a *compareTo*() en *Sorter.keepMoving*() para comparar dos objetos *Student*, se llamará polimórficamente a *OnCampusStudent.compareTo()* o a *OnlineStudent*.compareTo().

Además, estudia los comentarios del método *keepMoving*(), donde se ha utilizado y explicado cómo utilizar el **ternary oper-ator ?:** (heredado del lenguaje C). Hay una buena explicación de ?: en esta [página web](https://alvinalexander.com/java/edu/pj/pj010018).

## Clase de Estudiantes (Student)

En el archivo zip del proyecto hay una plantilla para la clase *Student*. La clase *Student* es una clase abstracta que implementa la interfaz *java.lang.Comparable*<T> (véase §5.3):

public abstract class Student implements Comparable<Student> { ... }

Un objeto *Student* contiene cinco variables de instancia (antepongo a los miembros de mis datos de instancia una **m** minúscula por **m**iembro):

*mCredits* Número de horas de crédito en las que está matriculado el estudiante.

*mFirstName* Nombre del(la) estudiante

*mId* El número de identificación del(la) estudiante.

*mLastName* Apellido del(la) estudiante.

*mTuititon* La matrícula computada del(la) estudiante.

Ten en cuenta que estos miembros de datos son comunes tanto a *OnCampusStudents* como a *OnlineStudents*. La mayoría de los métodos de instancia de *Student* deberían ser sencillos de implementar (la mayoría son simples accesos/métodos de mutador), así que sólo mencionaremos los dos que no son tan obvios:

+***calcTuition***(): void

Un método abstracto (por eso lo he escrito en cursiva aquí y en el diagrama de clases UML) que implementan las subclases de *Student*. Los métodos abstractos no suelen implementarse en una clase abstracta, y este no lo está. Para más información, consulta los comentarios de la cabecera del método *calcTuition* () en *Student*.

+compareTo(pStudent: Student): int «override»

Implementa el método *compareTo*() de la interfaz *Comparable*<*Student*>. Devuelve un entero negativo si la variable de instancia *mId* de *this* *Student* es menor que la variable de instancia *mId* de *pStudent*. Devuelve 0 si son iguales (no debería ocurrir porque los números de identificación son únicos). Devuelve un entero positivo si la variable de instancia *mId* de este *Student* es mayor que la variable de instancia *mId* de *pStudent*. El código de *compareTo*() es sencillo y se muestra a continuación. Lee los comentarios del método *compareTo* () en *Student* para obtener más información. Ten en cuenta que utilizarás la anotación @Override para evitar sobrecargas accidentales.

return getId().compareTo(pStudent.getId());

## Clase para estudiantes en el campus (OnCampusStudent)

La clase concreta *OnCampusStudent* es una subclase directa de la clase abstracta *Student*. Declara dos constantes *int* públicas *RESIDENT* que es 1 y *NON\_RESIDENT* que es 2. Se agregan nuevas variables de instancia que son específicas de los estudiantes en el campus:

*mResident RESIDENT* si el *OnCampusStudent* es residente, *NON\_RESIDENT* para no residente.

*mProgramFee* Algunos *OnCampusStudent*'s pagan una cuota de programa adicional. Este valor puede ser 0.

Los métodos de instancia de *OnCampusStudent* son, en su mayoría, sencillos de implementar, por lo que sólo hablaremos de dos de ellos.

+OnCampusStudent(pId: String, pFirstName: String, pLastName: String): «ctor»

Debe llamar al constructor de la superclase pasando *pId*, *pFirstName*, y *pLastName* como parámetros.

+calcTuition(): void «override»

Debe implementar las reglas descritas en §3 Antecedentes para calcular la matrícula de un estudiante residente o no residente. Ten en cuenta que no podemos acceder directamente a la variable de instancia *mTuition* de un *OnCampusStudent* porque está declarada intencionadamente como privada en *Student*. Entonces, ¿cómo escribimos en *mTuition*? Llamando al método mutador protegido *setTuition*() que se hereda de *Student*. ¿Por qué está protegido setTuition()? Porque sólo está pensado para ser llamado desde subclases de *Student* y no desde clases que no formen parte de la jerarquía de clases *Student*. El pseudocódigo de *calcTuition*() es:

Override Method calcTuititon() Returns Nothing Declare double variable t

If getResidency() returns RESIDENT Then t = TuitionConstants.ONCAMP\_RES\_BASE

Else

t = TuitionConstants.ONCAMP\_NONRES\_BASE End if

t = t + getProgramFee();

If getCredits() > TuitionConstants.ONCAMP\_MAX\_CREDITS Then

t = t + (getCredits() - TuitionConstants.ONCAMP\_MAX\_CREDITS) × TuitionConstants.ONCAMP\_ADD\_CREDITS

End if

Call setTuition(t) End Method calcTuition()

## Clase Estudiantes en línea (OnlineStudent)

La clase concreta *OnlineStudent* es una subclase directa de la clase abstracta *Student*. Añade una nueva variable de instancia específica para los estudiantes en línea:

La clase concreta *OnlineStudent* es una subclase directa de la clase abstracta *Student*. Se agrega una nueva variable de instancia específica para los estudiantes online:

*mTechFee* Algunos *OnlineStudent* pagan una tasa tecnológica adicional. Esta variable de instancia será verdadera si se aplica la tasa tecnológica y falsa si no se aplica.

Los métodos de instancia de *OnlineStudent* son, en su mayoría, sencillos de implementar, por lo que sólo hablaremos de dos de ellos.

+OnlineStudent(pId: String, pFirstName: String, pLastName: String): «ctor»

Debe llamar al constructor de la superclase pasando pId, *pFirstName* y *pLastName* como parámetros.

+calcTuition(): void «override»

Debe aplicar las reglas descritas en §3 Antecedentes. El pseudocódigo de *calcTuition*() es:

Override Method calcTuititon() Returns Nothing

Declare double variable t = getCredits() × TuitionConstants.ONLINE\_CREDIT\_RATE If getTechFee() returns true Then

t = t + TuitionConstants.ONLINE\_TECH\_FEE End if

Call setTuition(t) End Method calcTuition()

## Diagrama de clases UML

El diagrama de clases UML que se muestra en la Fig. 1 de la página siguiente se creó utilizando UMLet. Consulta la carpeta *cse205-p02/uml* del archivo zip para ver el archivo UMLet y la carpeta *cse205-p02/img* para ver las imágenes .EPS y .PNG del diagrama de clases. Tenemos las siguientes relaciones entre las clases y la interfaz *Comparable*<*Student*>. (Consulta esta [página web](https://www.uml-diagrams.org/class-reference.html) para ver un buen resumen de la notación utilizada en los diagramas de clases UML).

*Main*: Las líneas discontinuas con puntas de flecha abiertas que conectan *Main* con *Student*, *OnCampusStudent* y *OnlineStudent* representan relaciones de dependencia. *Main* depende de *Student* porque los objetos *Student* son parámetros de algunos métodos de *Main*, en concreto, *calcTuition*(), *readFile*() y *writeFile*(). Esto hace que *Main* dependa de *Student* porque si el código de *Student* cambia, podría afectar al código de *calcTuition*(), *readFile*() y *writeFile*. Observa que la punta de flecha apunta desde *Main* a la clase de la que depende *Main*. *Main* también tiene una línea continua sin símbolos en los extremos de las líneas que conectan con *Sorter*. Se trata de una relación de asociación. *Main* está asociada a *Sorter* porque *Main.run*() llama a *Sorter*. *insertionSort*(). Las asociaciones suelen incluir un texto que describe la asociación y yo he dibujado los **usos** del texto indicando que *Main* **utiliza** *la clase Sorter*.

*Student*: La línea continua con un símbolo de diamante sombreado que conecta a *Student* con *Main* representa una relación de composición. El objeto *Main* se compone de un *ArrayList* de objetos *Student*, véase *Main.run*(), y *Main* crea este objeto *ArrayList*, de modo que cuando el objeto *Main* es desasignado por el recolector de basura (podemos decir que *Main* muere) los objetos *Student* que estaban asignados en *Main* también mueren. Podemos decir que el ciclo de vida de *Main* y de los objetos *Student* de los que se compone *Main* es el mismo, por lo que se trata de una relación de composición y no de agregación. *Student* también tiene una línea de puntos que lo conecta a la interfaz *Comparable*<*Student*>. El **clasificador** UML "interfaz" sirve para indicar al lector que *Comparable*<*Student*> es una interfaz y no una clase. Una relación de interfaz se dibuja con una línea de puntos con un triángulo cerrado sin sombrear que apunta hacia la interfaz que se está implementando.

*Sorter*: La línea de puntos que conecta *Sorter* con *Student*, con una flecha abierta apuntando hacia *Student*, indica que *Sorter* depende de *Student*. *Sorter* depende de *Student* porque *Sorter.insertionSort*(), *Sorter.keepMoving*() y *Sorter.swap*() tienen métodos cuyos parámetros contienen objetos *Student*. Esto hace que *Sorter* dependa de *Student* porque si el código de *Student* cambia, podría requerir cambios en el código de *Sorter*.

*OnCampusStudent* y *OnlineStudent*: Las líneas continuas que conectan estas dos clases con *TuitionConstants* representan una relación de asociación y la palabra **usa (uses)** en cada línea simplemente indica al lector que estas dos clases utilizan la clase *Tuition*- *Constants*. Tanto *OnCampusStudent* como *OnlineStudent* tienen una línea continua con un triángulo sin sombrear que las conecta con *Student*. Estas líneas representan relaciones de generalización o herencia y, como el triángulo está en el extremo de la línea cerca de *Student*, indica que *Student* es la clase general y *OnCampusStudent* y *OnlineStudent* son las clases específicas. Una forma alternativa de describir esto es decir que *Student* es la superclase, y *OnCampusStudent* y *OnlineStudent* **heredan** de *Student* por lo que son subclases de *Student*.

# Requisitos adicionales del proyecto

1. Formatea tu código de forma ordenada. Utiliza la sangría y el espaciado adecuados. Estudia los ejemplos del libro y los ejemplos que el instructor presenta en las clases y publica en el sitio web del curso.
2. Coloca un bloque de encabezado de comentario en la parte superior de cada método formateado así:

/\*\*

\* ***A brief description of what the method does***.

\*/

1. Coloque un bloque de encabezado de comentario en la parte superior de cada archivo de código fuente con el siguiente formato:

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// CLASE: ***classname*** (***classname***.java)

//

// DESCRIPCIÓN

// ***Una descripción del contenido de este archivo***.

//

// INFORMACIÓN DEL CURSO Y DEL PROYECTO

// CSE205 Programación Orientada a Objetos y Estructuras de Datos, ***semestre y año***

// Número de Proyecto: ***numero-del-proyecto***

//

// AUTOR: ***su-nombre, su-asurite-id, su-email***

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Fig.1 Project 2 UML Class Diagram**

