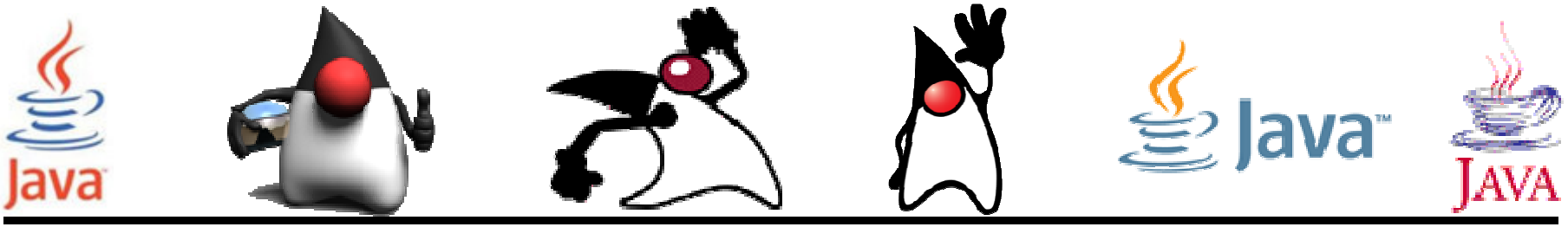


Creación de Clases

Pedro Corcuera

Dpto. Matemática Aplicada y
Ciencias de la Computación

Universidad de Cantabria

corcuerp@unican.es

Objetivos

- Creación de clases propias
- Declarar propiedades (fields) y métodos para las clases creadas
- Usar la referencia this para acceder a los datos instanciados
- Crear y llamar métodos sobrecargados
- Usar modificadores de acceso para controlar el acceso a los miembros de la clase
- Conocer los fundamentos del diseño de clases

Índice

- Objetos y Clases
- Modificadores de clase
- Declaración de miembros
- Modificadores de miembros
- Variables/métodos de instancia
- Variables/métodos de clase
- Sobrecarga de métodos
- Constructores
- Referencia null, this
- Diseño de clases

Objetos y Clases

- Para construir programas más grandes y complejos
- Para modelar objetos que existen en el mundo real
- En lo que sigue se usa para la sintaxis la notación

*	significa que puede haber 0 o más ocurrencias de la línea donde se aplica
<descripcion>	indica que se tiene que sustituir un nombre o valor para esta parte, en lugar de escribirlo como está
[ ]	indica que esta parte es opcional

Declaración de Clases propias

- La sintaxis para declarar una clase es:

```
<modificador> class <nombre>
    [extends SuperClase]
    [implements Interface1, Interface2  ...] {
    <declaracionAtributos>*
    <declaracionConstructor>*
    <declaracionMetodo>*
}
```

donde:

<modificador> es un modificador de acceso, que puede ser combinados con otros tipos de modificadores

Declaración de Clases propias - ejemplo

```
public class StudentRecord {
  // aqui se añade el código
}
```

donde:

`public` significa que la clase es accesible a otras clases externa del paquete

`class` es la palabra clave para crear una clase en Java

`StudentRecord` es un identificador único que describe la clase

Recomendaciones de codificación

- Poner un nombre apropiado a los nombres de las clases, que describa de manera compacta la entidad que se desea representar
- Los nombres de las clases deben empezar por letras mayúsculas
- El nombre del fichero que contiene la clase debe tener el mismo nombre de la clase (p.e. StudentRecord.java)

Modificadores de Clase

Modificador de clase	Efecto
`<nada>`	Cuando no se especifica un modificador, por defecto, la clase es accesible por todas las clases dentro del mismo paquete
`public`	Es accesible por cualquier clase
`abstract`	Contiene métodos abstractos
`final`	No se puede extender, esto es, no puede tener subclases

Declaración de atributos y métodos

- Sintaxis para declarar atributos o campos:

```
<modificador> <tipo> <name> [= <valor_inicial>];
```

- Sintaxis para declarar métodos:

```
<modificador><tipoRetorno><name>(<parameter>*) {
    <sentencias>*
}
```

donde:

<modificador> puede ser una combinación de modificadores

<tipoRetorno> puede ser cualquier tipo de dato (incluyendo void)

<name> puede ser cualquier identificador válido

<parameter> ::= <tipo_parameter> <name_parameter>[,]

Modificadores de métodos, campos y clases inner

Modificador miembro	Efecto
`<nada>`	Cuando no se especifica un modificador, por defecto, un miembro es accesible por todas las clases dentro del mismo paquete
`public`	El miembro es accesible por cualquier clase
`protected`	El miembro es accesible por la misma clase, todas las subclases y todas las clases dentro del mismo paquete
`private`	El miembro es accesible sólo por la misma clase
`static`	Un campo static es compartido por todas las instancias de la clase. Un método static sólo accede a campos estáticos
`final`	Un método final no puede ser sobreescrito en subclases. Un campo final tiene un valor constante que no se puede modificar

Modificadores en la declaración aplicados sólo a campos y métodos

Modificador de campo	Efecto
`volatile`	Un campo volatile puede ser modificado por métodos no sincronizados en un entorno multihilo
`transient`	Un miembro transient no es parte del estado persistente de las instancias

Modificador de método	Efecto
`abstract`	Un método abstracto difiere su implementación a las subclases
`sincronized`	Un método sincronizado es atómico en un entorno multihilo
`native`	Un método nativo es implementado en C y C++

Accesibilidad de miembros

Accesibilidad	Public	Protected	Package	Private
La misma clase	Si*	Si	Si	Si
Clases en el mismo package	Si	Si	Si	No^
Subclases en un package diferente	Si	Si	No	No
Ninguna subclase en un package diferente	Si	No	No	No

- Los modificadores public, protected o private sólo pueden aparecer en la lista de modificadores para cada miembro.

Variables de instancia

- Un objeto almacena datos en variables de instancia, que se declaran al comienzo de la clase
  - Variables declaradas dentro de la clase que son accedidas (acceso y modificación) por todos los métodos dentro de la clase
  - Si se declaran como miembros de datos `private` no se pueden acceder a ellas desde métodos fuera de la clase
  - Si se declaran `public` otras clases puedan tener acceso directo
  - Cada objeto de una clase tiene un conjunto separado de variables de instancia

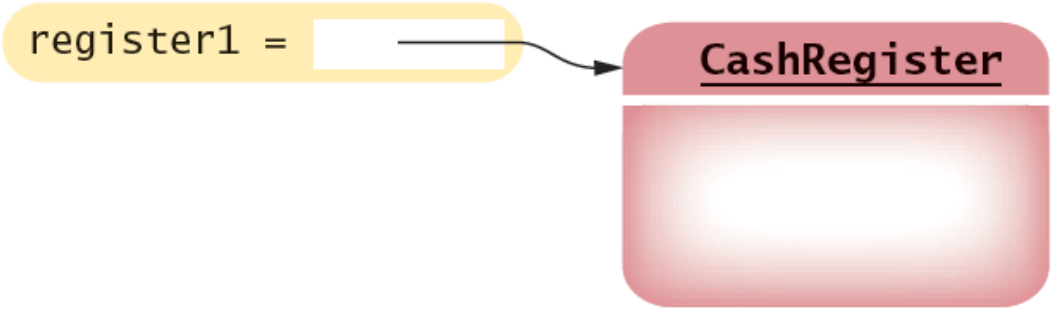
Métodos de instancia

- Para invocar métodos se debe instanciar un objeto
- En el diseño de una clase:
  - se especifican las tareas a realizar (interfaz pública)
  - qué métodos se necesitan, los parámetros y retorno de cada uno

Tareas de clase CashRegister	Método	Retorno
Añade precio de un artículo	`addItem(double)`	void
Obtiene la cantidad total comprada	`getTotal()`	double
Obtiene la cantidad de artículos	`getCount()`	int
Limpia la caja para una nueva venta	`clear()`	void

Métodos de instancia

- Creación del objeto



- Invocación de método

```
public void addItem(double val)
```

```
public static void main(String[] args) {
  // Crea un objeto CashRegister
  CashRegister register1 = new CashRegister();
  // Invoca un metodo de instancia del objeto
  register1.addItem(1.95);
}
```

Métodos accessor/mutator

- Muchos métodos caen en dos categorías:
  - Métodos accessor: '**get**'
    - usados para leer valores de variables (instancia/static)
    - devuelven un valor
    - Sintaxis: `get<NombreDeVariable>`

```
public String getName() { return name;  }
```

  - Métodos mutator: '**set**'
    - usados para escribir o cambiar valores de variables
    - normalmente devuelven void
    - Sintaxis: `set<NombreDeVariable>`

```
public void setName(String tmp) { name = tmp;  }
```

Variables de Instancia: Ejemplo

```
public class StudentRecord {
  // Variables de Instancia
  private String name; private String address;
  private int age;  private double mathGrade;
  private double englishGrade; private double scienceGrade;
  private double average;
  //se incluira mas codigo aqui
}
```

Cada variable de instancia se declara como cualquier otra variable usada hasta ahora

Cada objeto `StudentRecord` tiene una copia separada de las variables y los valores determinan el estado del objeto

```
public class StudentRecordExample {
  public static void main(String[] args) {
    // Crea un objeto de clase StudentRecord
    StudentRecord annaRecord =new StudentRecord();
    // Crea otro objeto de clase StudentRecord
    StudentRecord beahRecord =new StudentRecord();
    // Asigna valores a Anna y Beah
    annaRecord.setName("Anna");
    annaRecord.setEnglishGrade(95.5);
    beahRecord.setName("Beah“);
}
```

Creación de objetos e inicialización de valores usando métodos de acceso

Acceso a variables de instancia

- Las variables de instancia `private` no pueden ser accedidas desde métodos externos a la clase

```
public static void main(String[] args)
{ . . .
  System.out.println(annaRecord.average); // Error
  . . .
}
```

El compilador no permitirá la violación de privacidad

- Se usan métodos de acceso de la clase para ello

```
public static void main(String[] args)
{ . . .
  System.out.println(annaRecord.getName() ); // OK
  . . .
}
```

El encapsulado ofrece una interfaz pública y oculta los detalles de implementación

Recomendaciones de codificación

- Declarar todas las variables de instancia después de "public class MiClase {" y una variable por línea
- Las variables de instancia, como otras variables, deben empezar con una letra minúscula
- Usar un tipo de dato apropiado para cada variable que se declara
- Declarar las variables de instancia `private` para que sólo los métodos de la clase puedan acceder a ellas

Variables de clase (static)

- Se les conoce también como variables de clase. Se obtienen con `static` en la declaración

```
public class StudentRecord
{
  private String name;
  private static int studentCount;
  . . .
}
```

- Los métodos de cualquier objeto de la clase pueden usar o modificar el valor de la variable estática
- Para acceder a la variable estática: `NombreClase.nombreVariable`

Métodos estáticos (static)

- La API de Java tiene muchas clases que ofrecen métodos que se pueden usar sin instanciar objetos
  - La clase Math es un ejemplo ya utilizado
  - `Math.sqrt(value)` es un método estático que devuelve la raíz cuadrada de un valor
  - No es necesario instanciar la clase `Math`
- Para acceder a un método estático

```
ClassName.methodName()
```

Creación de métodos estáticos propios

- Los métodos estáticos normalmente retornan un valor

```
public class Financial {
  /** Calcula el porcentaje de una cantidad. */
  public static double percentOf(double percentage,
                                 double amount) {
    return (percentage / 100) * amount;
  }
}
```

- Invocación del método de la clase, no un objeto

```
double tax = Financial.percentOf(taxRate, total);
```

Recomendaciones de codificación

- Los nombres de métodos deben empezar con una letra minúscula y deben ser verbos
- Siempre escribir documentación antes de la declaración del método, preferentemente en formato Javadoc
- Cuándo definir métodos estáticos?
  - Cuando la lógica y estado no involucra la instancia específica de un objeto
  - Cuando la lógica es una conveniencia sin crear una instancia de un objeto

Ejemplo de variables y métodos en clases

```
public class StudentRecord {
    /** Creates a new instance of StudentRecord */
    public StudentRecord() { }
    private String name; private double mathGrade;
    private double englishGrade; private double scienceGrade;
    private static int studentCount = 0; // Declare static variables.
    /**  Returns the name of the student */
    public String getName(){ return name; }
    /** Changes the name of the student */
    public void setName(String temp ){ name =temp; }
    /** Computes the average of the english,math and science grades */
    public double getAverage(){
      double result =0;
      result =(getMathGrade()+getEnglishGrade()+getScienceGrade() )/3;
      return result;
    }
    /** Returns the number of instances of StudentRecords */
    public static int getStudentCount(){ return studentCount; }
}
```

Ejemplo de variables y métodos en clases

```
public class StudentRecordExample {
    /** Creates a new instance of StudentRecordExample */
    public static void main(String[] args) {
        StudentRecord annaRecord =new StudentRecord();
        StudentRecord.increaseStudentCount();
        StudentRecord beahRecord =new StudentRecord();
        StudentRecord.increaseStudentCount();
        StudentRecord crisRecord =new StudentRecord();
        StudentRecord.increaseStudentCount();
        // Set the names of the students.
        annaRecord.setName("Anna");
        beahRecord.setName("Beah");
        crisRecord.setName("Cris");
        // Print anna's name.
        System.out.println("Name = " + annaRecord.getName());
        // Print number of students.
        System.out.println("Student Count = "+
                           StudentRecord.getStudentCount());
     }
}
```

Sobrecarga (overloading) de métodos

- Los métodos sobrecargados tienen las siguientes propiedades:
  - Tienen el mismo nombre
  - Diferentes parámetros o número diferente de parámetros
  - Los tipos retornados pueden ser diferentes o los mismos
- Se usan cuando la misma operación tiene diferentes implementaciones
- La recomendación es: evitar la sobrecarga

Ejemplo de sobrecarga de métodos

```
// Overloaded myprint(..) methods
public void myprint(){
  System.out.println(“Version 1: Nada se pasa");
}

public void myprint(String name ){
  System.out.println(“Version 2: Name:"+name);
}

public void myprint(String name, double averageGrade){
  System.out.print(“Version 3: Name:"+name+" ");
  System.out.println("Average Grade:"+averageGrade);
}
```

Métodos constructores

- Un *constructor* es un método que inicializa las variables de instancia de un objeto.
- Las propiedades de un constructor son:
  - Los constructores tienen el mismo nombre que la clase
  - Un constructor se declara como otros métodos
  - Los constructores no retornan ningún valor, pero no se debe usar void en su declaración
  - No se llaman directamente. Se llaman automáticamente cuando se crea un objeto con new

Métodos constructores - declaración

- Sintaxis para declarar un constructor

```
<modificador> <nombreClase> (<parametro>*) {
  <sentencias>*
}
```

- Si no se especifica ningún constructor, el compilador crea un constructor *default* automáticamente
  - No tiene parámetros
  - Inicializa todas las variables de instancia

Métodos constructores - ejemplo

```
public class StudentRecord {
  //declaracion de variables instancia
  ...
  public StudentRecord() { // Default constructor }
  //Constructores con diferentes numeros de parametros
  public StudentRecord(String name)
  { this.name = name; }
  public StudentRecord(String name, double mGrade){
    this(name); mathGrade = mGrade; }
  public StudentRecord(String name, double mGrade,
                       double eGrade){
    this(name, mGrade); englishGrade = eGrade; }
  public StudentRecord(String name, double mGrade,
                       double eGrade, double sGrade){
    this(name, mGrade, eGrade);scienceGrade = sGrade;}
```

Métodos constructores - uso

```
public class ConstructorExample {
   public static void main(String[] args) {
   // Crea un objeto de la clase StudentRecord
   StudentRecord annaRecord = new
                                   StudentRecord("Anna");
  ...
  // Crea otro objeto de la clase StudentRecord
  StudentRecord beahRecord =
                         new StudentRecord("Beah", 45);
 ...
  // Crea otro objeto de la clase StudentRecord
  StudentRecord crisRecord =
              new StudentRecord("Cris", 23.3, 67.45, 56);
 ...
```

La referencia null

- Una referencia puede apuntar a un 'no' objeto
  - No se puede invocar métodos de un objeto mediante una referencia `null` - causa un error en tiempo de ejecución

```
CashRegister reg = null;
System.out.println(reg.getTotal());   // Error Runtime !
```

  - Para probar si una referencia es `null` antes de usarse:

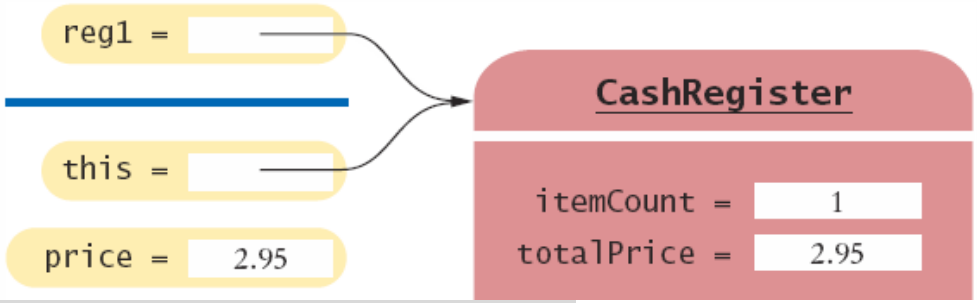
```
String middleInitial = null; // Sin segundo nombre

if (middleInitial == null)
  System.out.println(firstName + " " + lastName);
else
  System.out.println(firstName + " " + middleInitial + ". "
+ lastName);
```

La referencia `this`

- Los métodos reciben el 'parámetro implícito' en una variable referencia llamada `this`
  - Es una referencia al objeto con el que se invocó el método
  - Se usa para acceder a las variables de instancia y NO se puede usar con variables estáticas
  - Sintaxis: `this.<nombreVariableInstancia>`

```
void addItem(double price)
{
  this.itemCount++;
  this.totalPrice = this.totalPrice + price;
}
```



Llamada a Constructor `this()`

- Las llamadas a constructores se pueden encadenar: se puede llamar un constructor desde otro constructor
- Se puede usar `this()` para este propósito con las siguientes condiciones:
  - Debe ser la primera sentencia en el constructor
  - Sólo se puede usar en la definición de un constructor

```
public StudentRecord() { this("some string"); }
public StudentRecord(String temp) { this.name = temp; }
public static void main( String[] args ) {
   StudentRecord annaRecord = new StudentRecord();
}
```

Prueba de clases

- Cuando se implementa una clase es preciso verificar si el funcionamiento de la misma de forma aislada es correcto (prueba de unidad – **unit testing**)
- Se requiere desarrollar una clase probadora (método main) y los resultados esperados:
  - Deben probarse todos los métodos e imprimir los resultados y compararlos con los esperados
- Alternativamente, se puede usar herramientas automáticas como Junit (http://www.junit.org)

Pasos para implementar una clase

1. Listar (formato informal) las responsabilidades de los objetos
2. Especificar la interfaz pública
3. Documentar la interfaz pública (comentarios Javadoc)
4. Determinar las variables de instancia
5. Implementar los constructores y métodos
6. Probar la clase

Mostrar el menú
Obtener la entrada del usuario

```
public Menu();
public void addOption(String option);
public int getInput();
/** Añade una opcion al final del menu.
    @param option la opcion a añadir
*/
private ArrayList<String> options;


public void addOption(String option)
{
   options.add(option);
}
```

Descubriendo clases

- El diseño de un programa orientado a objetos es diferente del diseño de un programa estructurado
- Primero, hay que decidir qué clases usar
  - Clases creadas por uno mismo
  - Reutilizar clases existentes
- Examinar cuidadosamente la especificación del problema para hallar las clases que se necesitan
  - Buscar nombres
  - Buscar conceptos
  - Los métodos vienen asociados por los verbos o acciones

Programas con múltiples clases

- Los programas normalmente usan varias clases en los que existe una relación entre ellas
- Una de las relaciones fundamentales entre clases es la "*agregación*"
  - se conoce informalmente como una relación "tiene-un"
- En UML la relación de agregación en el diagrama de clases se representa con