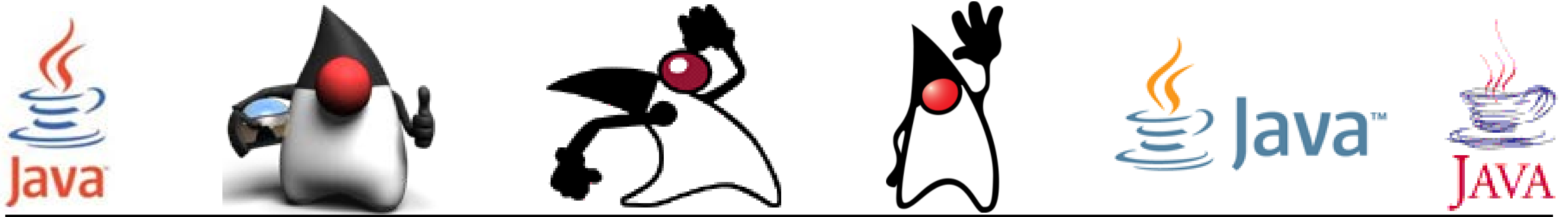


Creación de Clases

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

Clases Wrapper

- Los tipos de datos primitivos no son objetos y por tanto no pueden acceder a los métodos de la clase *Object*
- Las clases Wrapper permiten la representación en objetos de los tipos primitivos

Clases Wrapper

Tipo primitivo	Clase Wrapper	Argumentos Constructor
byte	Byte	byte o String
short	Short	short o String
int	Integer	int o String
long	Long	long o String
float	Float	float, double o String
double	Double	double o String
char	Character	char
boolean	Boolean	boolean o String

Clase Wrapper Boolean: Ejemplo

```
class BooleanWrapper {
  public static void main(String args[]) {
    boolean booleanVar = 1 > 2;
    Boolean booleanObj = new Boolean("Ttue");
    /* primitivo a objeto; tambien se puede usar metodo valueOf */
    Boolean booleanObj2 = new Boolean(booleanVar);

    System.out.println("booleanVar = " + booleanVar);
    System.out.println("booleanObj = " + booleanObj);
    System.out.println("booleanObj2 = " +  booleanObj2);
    System.out.println("compare 2 wrapper objects: "
                     + booleanObj.equals(booleanObj2));

    /* objeto a primitivo */
    booleanVar = booleanObj.booleanValue();
    System.out.println("booleanVar = " + booleanVar);
  }
}
```

Clases Wrapper

Integer	clase wrapper del tipo primitivo int
Double	clase wrapper del tipo primitivo double
Long	clase wrapper del tipo primitivo long

- Ejemplos:

```
// crea instancia de la clase Integer con valor 7801
Integer dataCount = new Integer(7801);
// convierte un objeto Integer al tipo int
int newCount = dataCount.intValue();
// convierte un String al tipo int
String cants = "65000";
int canti = Integer.parseInt(cants);
```

Clase String: Ejemplo$_1$

```
class StringDemo {
  public static void main(String args[]) {
    String name = "Juan";
    System.out.println("name: " + name);
    System.out.println("3r caracter de name: " + name.charAt(2));
    // el caracter primero alfabeticamente tiene menor valor unicode
    System.out.println("comp Juan con Sara: "+ name.compareTo("Sara"));
    System.out.println("cm Sara con Juan: "+ "Sara".compareTo("Juan"));
    // 'J' tiene menor valor unicode comparado a 'j'
    System.out.println("cm Juan con juan: " + name.compareTo("juan"));
    System.out.println("Juan comparado" +
                        name.compareToIgnoreCase("juan") );
    System.out.println("es Juan igual a Juan? " + name.equals("Juan"));
    System.out.println("es Juan igual a juan? " + name.equals("juan"));
    System.out.println("es Juan igual a juan (ignorar may/min)? " +
                        name.equalsIgnoreCase("juan"));
    char charArr[] = "Hi XX".toCharArray();
    "Juan".getChars(0, 2, charArr, 3);
    System.out.print("getChars metodo: ");
```

Clase String: Ejemplo$_2$

```
    System.out.println(charArr);
    System.out.println("Long. de name: " + name.length());
    System.out.println("Remplaza a's con e's en name: " +
                       name.replace('a', 'e'));
    // se requiere añadir 1 al parametro endIndex de substring
    System.out.println("substring de name: " + name.substring(0, 2));
    System.out.println("Trim \"  a b c d e f   \": \"" +
                      "  a b c d e f   ".trim() + "\"");
    System.out.println("represent String de expresion boolean 10>10: "
                      + String.valueOf(10 > 10));
    // metodo toString se llama en el metodo println
    System.out.println("represent String de boolean expression 10<10: "
                      + (10<10));
    // Notar que no hay cambio en el nombre del objeto String
    System.out.println("name: " + name);
  }
}
```

Clase Date

- Representa un momento preciso de tiempo a nivel de milisegundos
- Los datos de fecha se representan como un tipo long que cuenta el número de milisegundos desde el 1 de Enero 1970 (Greenwich)
- Documentación Java SE 6 API

  http://download.oracle.com/javase/6/docs/api/java/util/Date.html

Clase Date: Ejemplo

```
import java.util.Date;

public class DateTest {
  public static void main(String[] args) {
     // Imprime el numero de milisegundos que toma
     // la ejecucion de un trozo de codigo
     Date d1 = new Date();

     // codigo a ser medido
     for (int i=0; i<10000000; i++) {
          int j = i;
     }

     Date d2 = new Date();
     long elapsed_time = d2.getTime() - d1.getTime();

     System.out.println("Duracion " +elapsed_time+" milisegundos");
  }
}
```

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

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

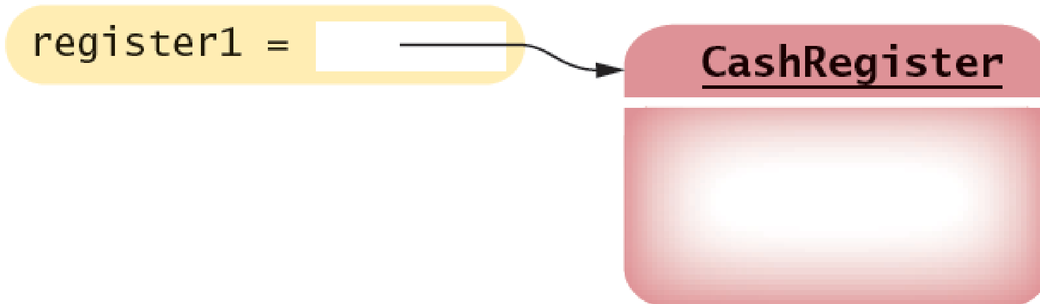
Accesibilidad	Public	Protected	Package	Private
La misma clase	Si*	Si	Si	Si
Clases en el mismo package	Si	Si	Si	No^
Subclases en un package diferente	Si	Si	No	No
Ninguna subclase en un package diferente	Si	No	No	No

*Accesible ^No accesible

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

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

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`

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 *constructo*r 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 !
```

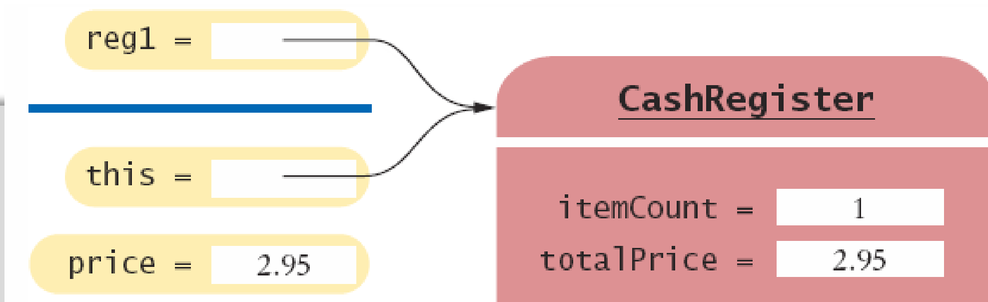
  - Para probar si una referencia es null antes de usarse:

```
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 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();
}
```

Llamada a método toString

- El método `toString` retorna la representación String de un objeto
- La clase Rectangle (java.awt) tiene un método toString
  - Se puede invocar directamente

```
Rectangle box = new Rectangle(5, 10, 20, 30);
String s = box.toString();   // Invoca toString directamente
// Asigna s a "java.awt.Rectangle[x=5,y=10,width=20,height=30]"
```

  - Se invoca implicítamente cuando se concatena un String con un objeto

```
System.out.println("box=" + box); // Llamada toString implicito
```

Sobreescribiendo el método toString

- Para una clase creada se puede usar `toString` sin sobreescribir

```
CuentaBanco miAhorro = new CuentaBanco(5000);
String s = miAhorro.toString();
// Asigna a s algo como “CuentaBanco@d24606bf"
```

  – Contiene el nombre de la clase seguido del código hash del objeto (no es muy útil)

- Si se sobreescribe, se puede obtener el estado del objeto

```
public class CuentaBanco {
  public String toString() {
// retorna “CuentaBanco[balance=5000]"
    return "CuentaBanco[balance=" + balance + "]";}  }
```

Clases final

- Son clases que no se pueden extender. Declaración:

```
public final ClassName {
...
}
```

- Ejemplos de clases finales son las clases wrapper y la clase String

Métodos final

- Son métodos que no se pueden sobreescribir. Declaración:

```
public final
[tipoReturn][nombreMet]([parametros]) {
 ...
 }
```

- Los métodos static son automáticamente final

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

Diagramas de clase

	Student	Circle
Name (Identifier)	**Student**	**Circle**
Variables (Static attributes)	name grade	radius color
Methods (Dynamic behaviors)	getName() printGrade()	getRadius() getArea()

SoccerPlayer	Car
name number xLocation yLocation	plateNumber xLocation yLocation speed
run() jump() kickBall()	move() park() accelerate()

Examples of classes

- **UML Notation**: In UML class diagram, public members are denoted with a "+"; while private members with a "-".

Ejemplo de clase

Class Definition

Circle
-radius:double=1.0 -color:String="red"
+getRadius():double +getColor():String +getArea():double

Instances

c1:Circle	c2:Circle	c3:Circle
-radius=2.0 -color="blue"	-radius=2.0 -color="red"	-radius=1.0 -color="red"
+getRadius() +getColor() +getArea()	+getRadius() +getColor() +getArea()	+getRadius() +getColor() +getArea()

Clase Circle

Circle
-radius:double = 1.0 -color:String = "red"
+Circle(radius:double,color:String) +Circle(radius:double) +Circle() +getRadius():double +setRadius(radius:double):void +getColor():String +setColor(color:String):void +getArea():double +toString():String

Circle.java

```
public class Circle {    // Save as "Circle.java"
   // Private instance variables
   private double radius;
   private String color;
   // Constructors (overloaded)
   public Circle() {                    // 1st Constructor
      radius = 1.0;
      color = "red";
   }
   public Circle(double r) {            // 2nd Constructor
      radius = r;
      color = "red";
   }
   public Circle(double r, String c) { // 3rd Constructor
      radius = r;
      color = c;
   }
   // Public methods
   public double getRadius() {
      return radius;
   }
   public String getColor() {
      return color;
   }
   public double getArea() {
      return radius * radius * Math.PI;
   }
}
```

TestCircle.java

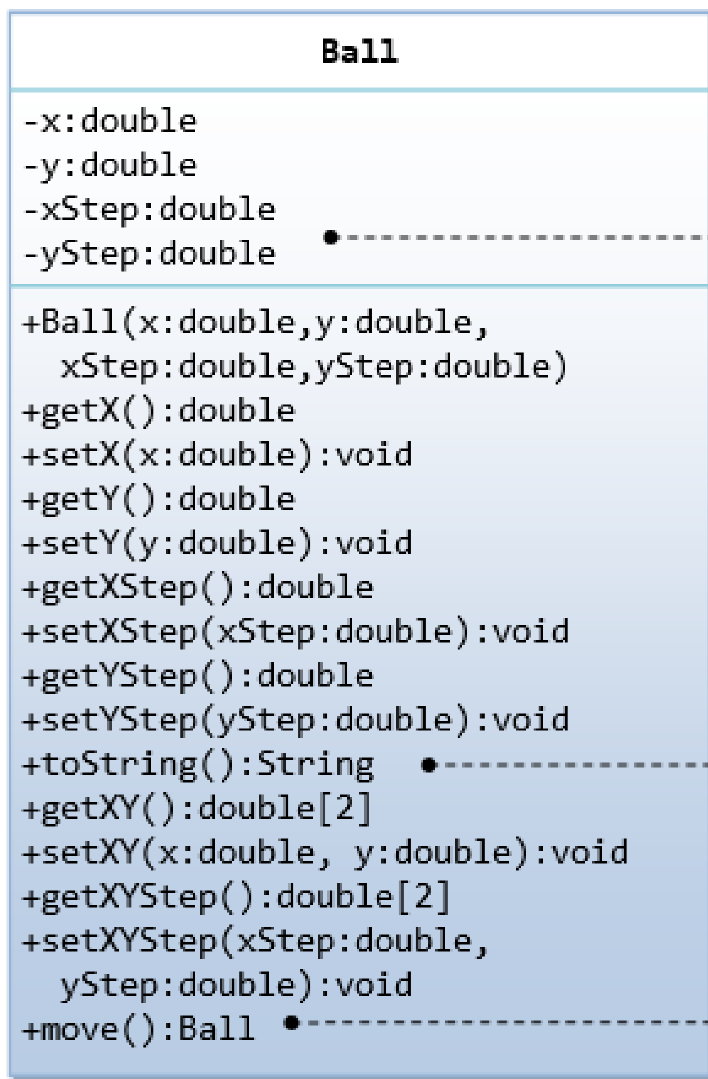
```
public class TestCircle {    // Save as "TestCircle.java"
   public static void main(String[] args) {   // Program entry point
      // Declare and Construct an instance of the Circle class called c1
      Circle c1 = new Circle(2.0, "blue");  // Use 3rd constructor
      System.out.println("The radius is: " + c1.getRadius()); // use dot
      System.out.println("The color is: " + c1.getColor());
      System.out.printf("The area is: %.2f%n", c1.getArea());
      // Declare and Construct instance of the Circle class called c2
      Circle c2 = new Circle(2.0);  // Use 2nd constructor
      System.out.println("The radius is: " + c2.getRadius());
      System.out.println("The color is: " + c2.getColor());
      System.out.printf("The area is: %.2f%n", c2.getArea());
      // Declare and Construct another instance called c3
      Circle c3 = new Circle();  // Use 1st constructor
      System.out.println("The radius is: " + c3.getRadius());
      System.out.println("The color is: " + c3.getColor());
      System.out.printf("The area is: %.2f%n", c3.getArea());
   }
}
```

Clase Point

Point
-x:int = 0 -y:int = 0
+Point() +Point(x:int, y:int) +getX():int +setX(x:int):void +getY():int +setY(y:int):void +toString():String +getXY():int[2] +setXY(x:int, y:int):void +distance(x:int,y:int):double +distance(another:Point):double +distance():double

- +toString():String — "(x,y)"
- +getXY():int[2] — Return a 2-element int array of {x,y}
- +distance(x:int,y:int):double — Return the distance from this instance to the given (x,y)
- +distance(another:Point):double — Return the distance from this instance to the given Point instance another
- +distance():double — Return the distance from this to (0,0)

Clase Ball



Displacement per move step

"Ball@(x,y),Speed=(xStep,yStep)"

Move one step.
x += xStep; y += yStep;
Return this instance.

Clase Student

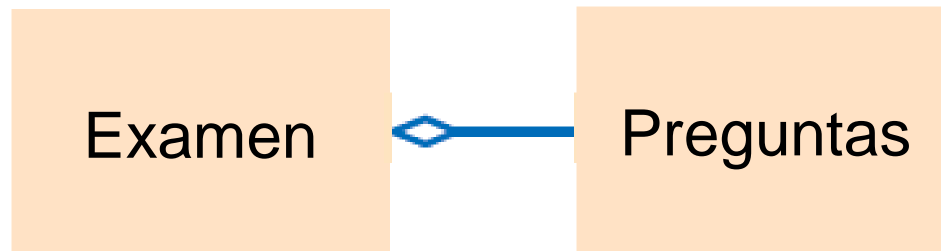
```
Student
-name:String
-address:String
-numCourses:int = 0
-courses:String[30] = {}
-grades:int[30] = {}
+Student(name:String, address:String)
+getName():String
+getAddress():String
+setAddress(address:String):void
+toString():String
+addCourseGrade(course:String,grade:int):void
+printGrades():void
+getAverageGrade():double
```

"name(address)"

"name course1:grade1, course2:grade2,..."

Programas con múltiples clases - Composición

- Los programas usualmente 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



Clase Author

Author
-name:String -email:String -gender:char
+Author(name:String,email:String, gender:char) +getName():String +getEmail():String +setEmail(email:String):void +getGender():char +toString():String

Clase Book

```
Book
-name:String
-author:Author
-price:double
-qty:int
+Book(name:String, author:Author,
   price:double, qty:int)
+getName():String
+getAuthor():Author
+getPrice():double
+setPrice(price:double):void
+getQty():int
+setQty(qty:int):void
+toString():String
```

1

```
Author
-name:String
-email:String
-gender:char
```

"'*book-name*' by *author-name* (*gender*) at *email*"

- **UML Notation**: In UML notations, composition is represented as a diamond-head line pointing to its constituents.

Clase Point

Point
-x:int = 0 -y:int = 0
+Point() +Point(x:int, y:int) +getX():int +setX(x:int):void +getY():int +setY(y:int):void +toString():String +getXY():int[2] +setXY(x:int, y:int):void +distance(x:int,y:int):double +distance(another:Point):double +distance():double

- toString(): "(x,y)"
- getXY(): Return a 2-element int array of {x,y}
- distance(x:int,y:int): Return the distance from this instance to the given (x,y)
- distance(another:Point): Return the distance from this instance to the given Point instance another
- distance(): Return the distance from this to (0,0)

Clase Line

Line
-begin:Point -end:Point
+Line(x1:int,y1:int,x2:int,y2:int) +Line(begin:Point,end:Point) +getBegin():Point +setBegin(begin:Point):void +getEnd():Point +setEnd(end:Point):void +getBeginX():int +setBeginX(x:int):void +getBeginY():int +setBeginY(y:int):void +getBeginXY():int[2] +setBeginXY(x:int,y:int):void +getEndX():int +setEndX(x:int):void +getEndY():int +setEndY(y:int):void +getEndXY():int[] +setEndXY(x:int,y:int):void +toString():String +getLength():double

1 — 2

Point
-x:int -y:int

"Line[begin=(x1,y1),end=(x2,y2)]"

Clase Circle

```
Circle
-center:Point = {0, 0}
-radius:double = 1.0
+Circle()
+Circle(xCenter:int,yCenter:int,
   radius:double)
+Circle(center:Point,
   radius:double)
+getRadius():double
+setRadius(radius:double):void
+getCenter():Point
+setCenter(center:Point):void
+getCenterX():int
+setCenterX(x:int):void
+getCenterY():int
+setCenterY(y:int):void
+getCenterXY():int[2]
+setCenterXY(x:int,y:int):void
+toString():String
+getArea():double
+getCircumference():double
+distance(another:Circle):double
```

1

```
Point
-x:int
-y:int
```

"Circle[center=(*x*,*y*),radius=*r*]"

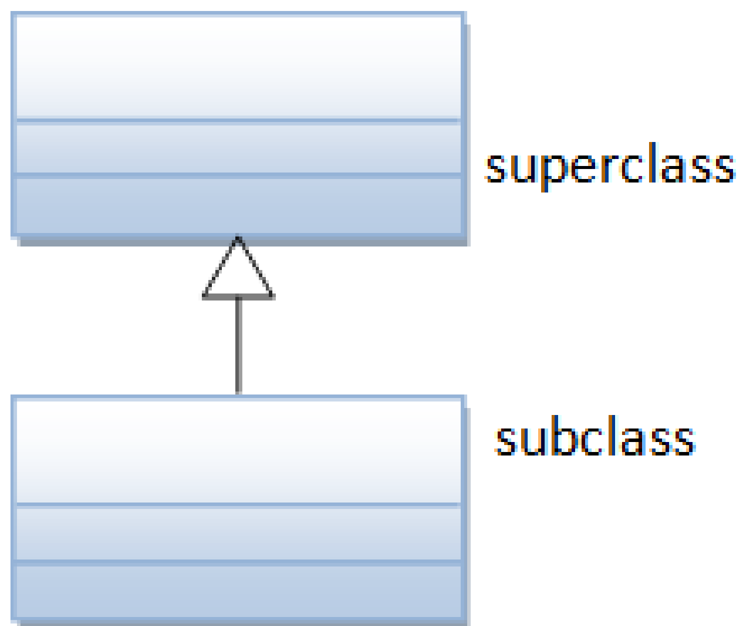
¿Qué es herencia?

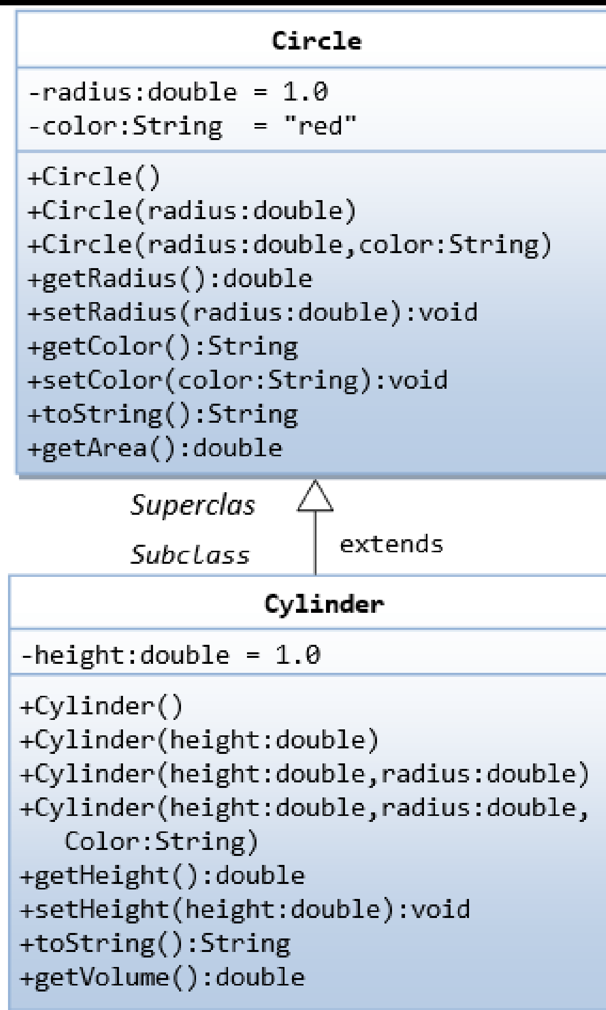
- Característica primaria de la programación orientada a objetos
- La herencia es una relación entre
  - una *superclase*: clase más general
  - una *subclase*: clase más especializada
- La subclase 'hereda' los datos (variables) y comportamientos (métodos) de la superclase

La Clase Object

- La clase Object es la madre de todas las subclases
  - En lenguaje Java, todas las clases son derivadas o extendidas a partir de la super clase Object
  - La clase Object es la única que no tiene una clase padre
- La clase Object define e implementa el comportamiento común a todas las clases incluyendo las que uno escribe
  - getClass()
  - equals()
  - toString() …

Herencia





- **UML Notation**: The UML notation for inheritance is a solid line with a hollow arrowhead leading from the subclass to its superclass. By convention, superclass is drawn on top of its subclasses as shown.

Palabra reservada super

- Una subclase puede llamar *explícitamente* un constructor de su superclase inmediata
- Esto se consigue por el uso de super para la llamada al constructor
- Una llamada del constructor super() en el constructor de una subclase resultará en la ejecución del constructor de la superclase basado en los argumentos pasados
  - La llamada a super() debe ser la primera sentencia en el constructor y sólo se puede usar en un constructor

Palabra reservada super

- Otro uso de super es para referirse a miembros de la superclase (como la referencia this)
- Ejemplos de uso de super:

```
public class Student extends Person {
  public Student(){
    super("UnNombre", "UnaDireccion");
    System.out.println("Inside Student:Constructor");
    super.name = "UnNombre";
    super.address = "UnaDireccion";
  }
  . . . .
}
```

Sobreescritura (overriding) de métodos

- Si una clase derivada necesita tener una versión diferente de la instancia de un método de la superclase, sobreescribe la instancia del método en la subclase
- El método que sobreescribe tiene el mismo nombre, número y tipo de parámetros y tipo de retorno que el método sobreescrito
- El método que sobreescribe puede también retornar un subtipo del tipo retornado por el método sobreescrito. Al tipo retornado se le llama covariante

Ejemplo de (overriding) de métodos

```
public class Person {
  :
  public String getName() {
    System.out.println("Parent: getName");
    return name;
  }
}
-----------------------------------------------
public class Student extends Person {
  :
  public String getName() {
    System.out.println("Student:getName");
    return name;
  }
  :
}
```

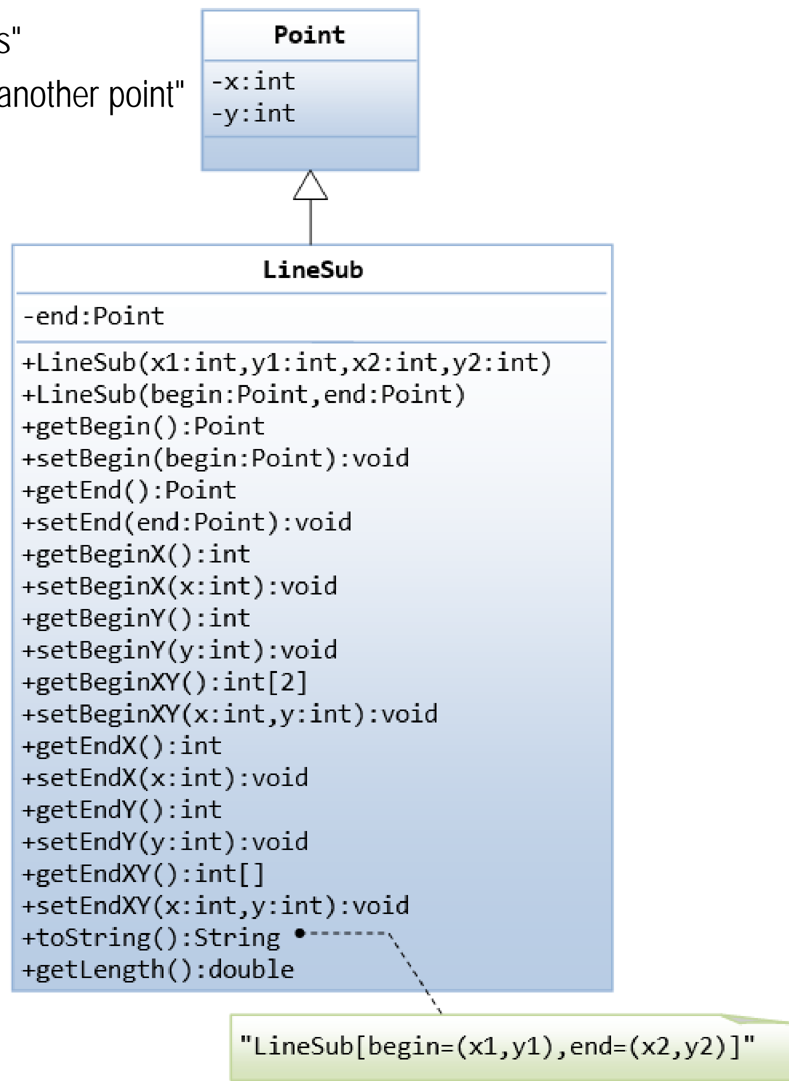
Cuando se invoca el método getName de un objeto Student, el resultado es: `Student:getName`

Herencia

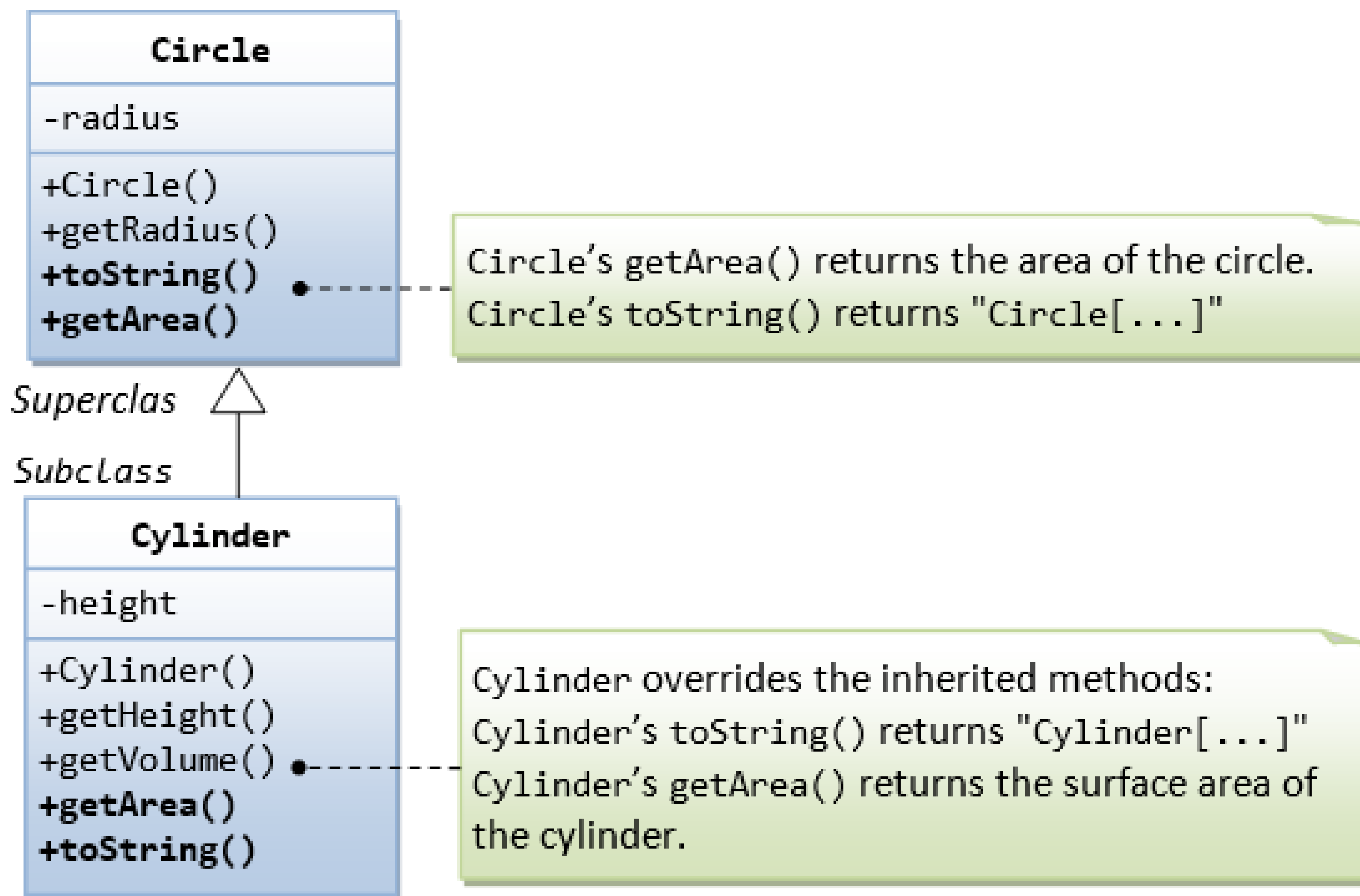
Herencia

Composición vs. Herencia

- "A line is composed of 2 points"
- "A line is a point extended by another point"



Polimorfismo



Polimorfismo

Polimorfismo

Monster
-name:String
+attack():String

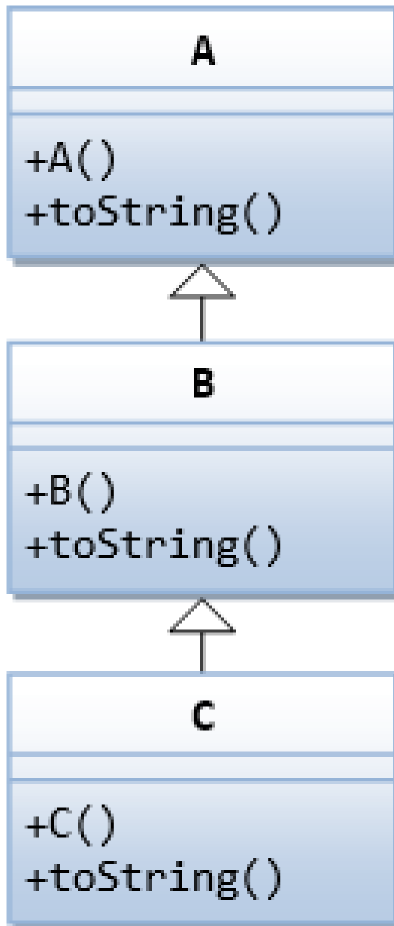
Superclass defines the expected behaviors (public interface) of all subclasses.
Program at the public interface.

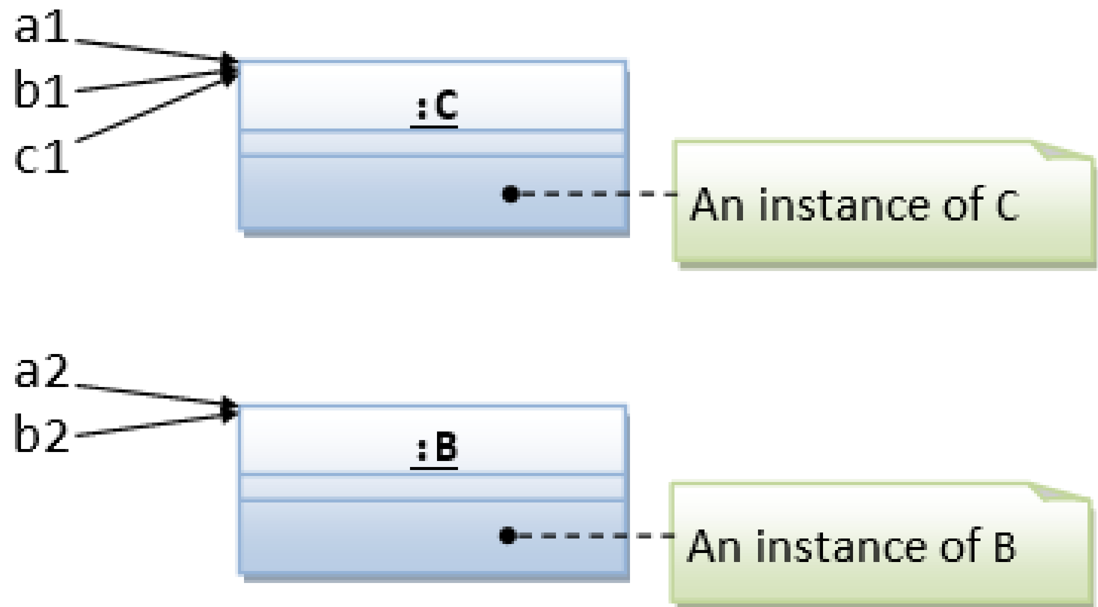
FireMonster	WaterMonster	StoneMonster
+attack():String	+attack():String	+attack():String

Subclasses provide the actual Implementations.

Upcasting y Downcasting

- Substituting a subclass instance for its superclass is called "upcasting"
- Downcasting requires explicit type casting operator in the form of prefix operator (new-type)





Método abstracto

- An abstract method is a method with only signature (i.e., the method name, the list of arguments and the return type) without implementation (i.e., the method's body).
- You use the keyword abstract to declare an abstract method.

```
abstract public class Shape {
   ......
   public abstract double getArea();
   public abstract void draw();
}
```

Clase abstracta

- A class containing one or more abstract methods is called an abstract class. An abstract class must be declared with a class-modifier **abstract**.

```
abstract public class Shape {
   private String color; // Private member variable
   // Constructor
   public Shape (String color) { this.color = color;}
   @Override
   public String toString() {
      return "Shape of color=\"" + color + "\"";
   }
   // All Shape subclasses must implement method getArea()
   abstract public double getArea();
}
```

Clases Abstractas

Shape

-color:String

+getArea():double
+toString():String

The abstract class and method are shown in *italic*

An abstract method has definition only

Rectangle

-length:int
-width:int

+getArea():double
+toString():String

Triangle

-base:int
-height:int

+getArea():double
+toString():String

Subclasses provide actual Implementation

- **UML Notation**: abstract class and method are shown in italic.

Interfaces

- An interface contains only public abstract methods (methods with signature and no implementation) and possibly constants (public static final variables).
- You have to use the keyword "interface" to define an interface.

```
public interface Movable {
   // abstract methods to be implemented by the subclasses
   public void moveUp();
   public void moveDown();
   public void moveLeft();
   public void moveRight();
}
```

Interfaces

- An interface is a contract (or a protocol, or a common understanding) of what the classes can do.
- Interface defines a set of common behaviors. The classes provide their own implementation.
- Java does not support multiple inheritance (whereas C++ does). Instead, Java does this by permitting you to "implements" more than one interfaces.

```
public class Circle extends Shape implements Movable, Displayable
{  // One superclass but implement multiple interfaces
   .......
}
```

Interface

Shape

+getArea():double

implements

Rectangle

-length:double
-width:double

+getArea():double
+toString():String

Triangle

-base:double
-height:double

+getArea():double
+toString():String

Interface defines the expected behaviors (public interface) of all

Subclasses provide the actual Implementations.

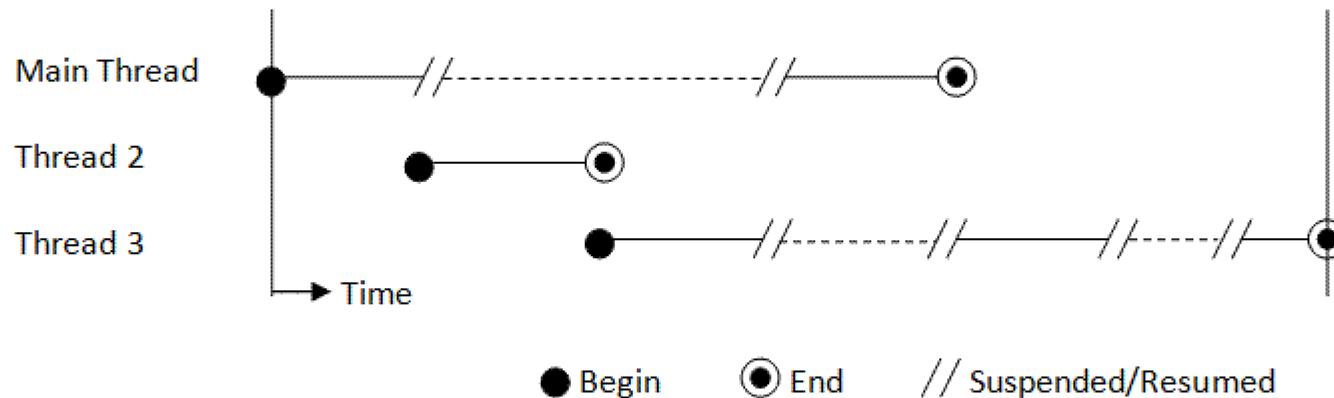
- **UML Notations**: Abstract classes, Interfaces and abstract methods are shown in italics. Implementation of interface is marked by a dash-arrow leading from the subclasses to the interface.

Notación UML Clases, Abstract,Interface

An Ordinary Superclass	An Abstract Superclass	An Interface
AConcreteClass	***AnAbstractClass***	***AnInterface***
anMethod()	*anAbstractMethod()* — Definition	*anAbstractMethod()* — Definition
extends	extends	implements
ASubClass	**ASubClass**	**ASubClass**
anMethod() — Override	anAbstractMethod() — Provide actual implementation	anAbstractMethod() — Provide actual implementation

Threads

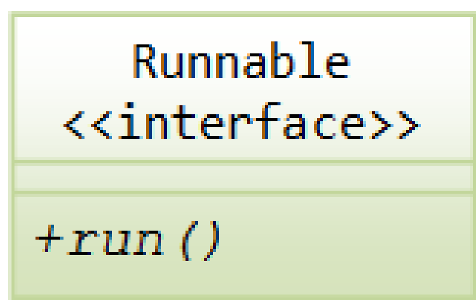
- A thread (lightweight process) is a single sequential flow of programming operations, with a definite beginning and an end.
- Java has built-in support for concurrent programming by running multiple threads concurrently within a single program.

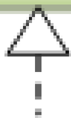


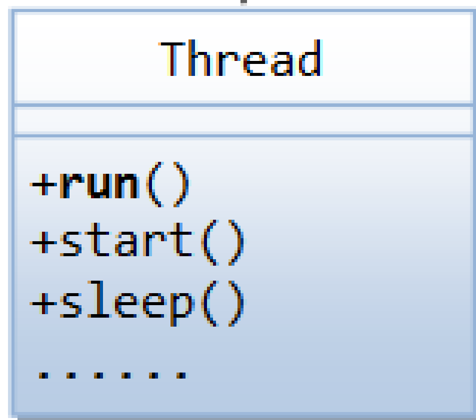
Creación de Threads

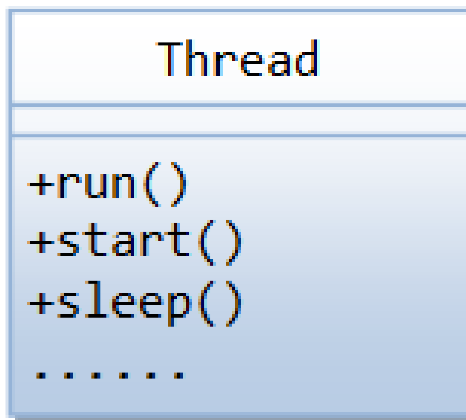
- There are two ways to create a new thread:
  - Extend a subclass from the superclass Thread and override the run() method to specify the running behavior of the thread. Create an instance and invoke the start() method, which will call-back the run() on a new thread.
  - Create a class that implements the Runnable interface and provide the implementation to the abstract method run() to specify the running behavior of the thread. Construct a new Thread instance using the constructor with a Runnable object and invoke the start() method, which will call back run() on a new thread.

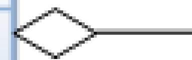
Clase Thread

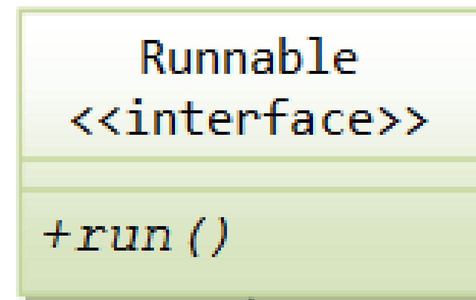


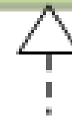


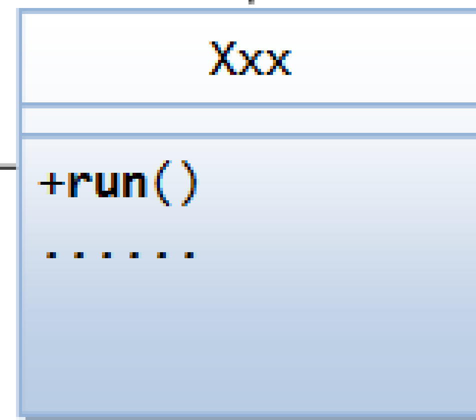












Qué es una excepción?

- Evento excepcional, típicamente un error que ocurre en tiempo de ejecución
- Provoca una interrupción en la ejecución normal de un programa
- Ejemplos:
  - operaciones de división por cero
  - acceder a elementos de una array fuera de rango
  - entrada inválida
  - apertura de un fichero inexistente
  - agotamiento de memoria

Captura de excepciones: sentencia try-catch

- Sintaxis para capturar excepciones con try-catch:

```
try {
   <codigo a monitorizar por excepciones>
} catch (<ExceptionType1> <ObjName>) {
   <manejador si ocurre ExceptionType1 >
}
...
} catch (<ExceptionTypeN> <ObjName>) {
   <manejador si ocurre ExceptionTypeN >
}
```

Ejemplos: captura de excepciones con try-catch

```
class DivByZero {
  public static void main(String args[]) {
    try {
      System.out.println(3/0);
      System.out.println("Imprime esto pf.");
    } catch (ArithmeticException exc) {
      //Division por cero es una ArithmeticException
      System.out.println(exc);
    }
    System.out.println("Despues excepcion.");
  }
}
```

Ejemplos: captura de excepciones con try-catch

```
class MultipleCatch {
  public static void main(String args[]) {
    try {
      int den = Integer.parseInt(args[0]);
      System.out.println(3/den);
    } catch (ArithmeticException exc) {
      System.out.println("Divisor 0");
    } catch (ArrayIndexOutOfBoundsException exc2) {
      System.out.println("Fuera de rango");
    }
    System.out.println("Despues excepcion");
  }
}
```

Ejemplos: captura de excepciones con try anidados

```
class NestedTryDemo {
  public static void main(String args[]){
    try {
      int a = Integer.parseInt(args[0]);
        try {
          int b = Integer.parseInt(args[1]);
          System.out.println(a/b);
        } catch (ArithmeticException e) {
          System.out.println("Division por cero!");
        }
    } catch (ArrayIndexOutOfBoundsException) {
      System.out.println("Requiere 2 parametros");
    }
  }
}
```

Ejemplos: captura de excepciones con try anidados con métodos

```
class NestedTryDemo2 {
  static void nestedTry(String args[]) {
    try {
      int a = Integer.parseInt(args[0]);
      int b = Integer.parseInt(args[1]);
      System.out.println(a/b);
    } catch (ArithmeticException e) {
      System.out.println("Division por cero!");
    }
  }
  public static void main(String args[]){
    try {
      nestedTry(args);
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Requiere dos parametros");
    }
  }
}
```

Captura de excepciones: palabra reservada *finally*

- Contiene código para "limpieza" después de un try-catch. Se ejecuta independientemente de errores
- Sintaxis try-catch-finally:

```
try {
  <codigo a monitorizar por excepciones>
} catch (<ExceptionType1> <ObjName>) {
  <manejador si ocurre ExceptionType1 >
} ...
} finally {
  <codigo a ejecutar antes termine bloque try>
}
```