IN77J – Orientación al Objeto para el e-business

2. Fundamentos de la OOP

Temario

- 2. Fundamentos de la OOP
  - Principios Fundamentales de la OOP
  - ¿Por qué Usar OOP?
  - Clases y Objetos
  - Encapsulación
  - Herencia
  - Polimorfismo
  - Ejemplo de Diseño OO

OOP

- En Object-Oriented Programming (OOP), o Programación Orientada a Objetos, un programa está hecho de clases, con sus propiedades y operaciones

- La creación de un programa involucra ensamblar objetos y hacerlos comunicarse entre ellos (componer)

OOP vs Programación Procedural

- En la programación procedural, un sistema se descompone en estructuras de datos y rutinas
- En OOP un sistema se descompone en clases, que se estructuran jerárquicamente usando herencia
- En un sistema OO puro, las variables y los métodos residen al interior de las clases
- Las clases permiten ocultar parte de su implementación (encapsulación)
- Objetos de diferentes clases pueden manejarse a través de interfaces comunes (polimorfismo)
- Las clases se agrupan en módulos (llamados paquetes en Java, namespaces en .NET)

¿Por Qué Usar OOP?

- Mayor modularidad: la posibilidad de ocultar la implementación (encapsulación), así como la posibilidad de ver diferentes clases a través de una interfaz común (polimorfismo), brindan mayor modularidad al código
- La herencia y el polimorfismo entregan facilidades para construir código reutilizable que no existen en la programación procedural
- El hecho de que haya un mayor nivel de reutilización reduce el costo de mantenimiento
- El ocultamiento de la implementación también ayuda a reducir el costo de mantenimiento (permite modificar la implementación sin impactar a los clientes)

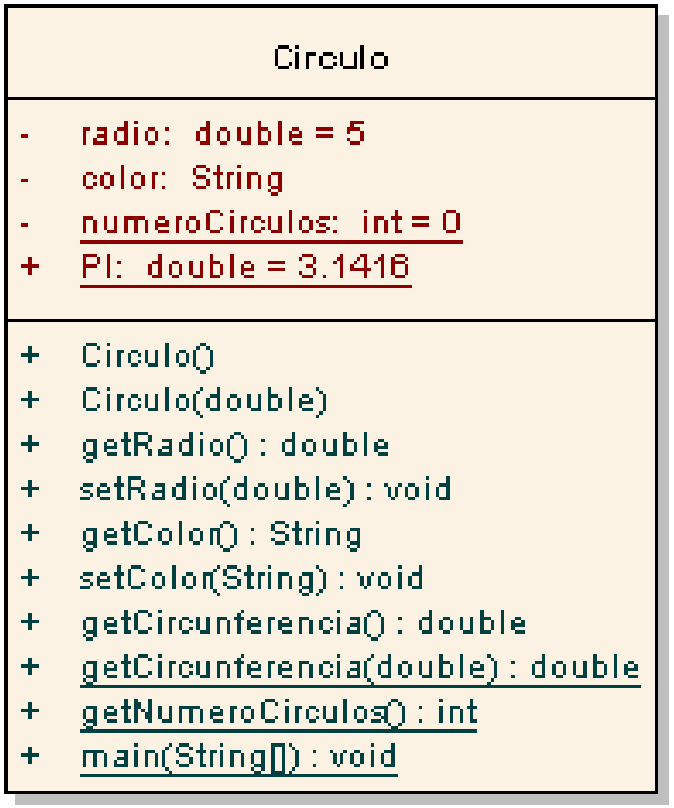
Clases y Objetos

- Una clase describe un grupo de objetos que comparten propiedades y métodos comunes
- Una clase es una plantilla que define qué forma tienen los objetos de la clase
- Una clase se compone de:
  - Información: campos (atributos, propiedades)
  - Comportamiento: métodos (operaciones, funciones)
- Un objeto es una instancia de una clase

Clase	**Objeto**
Empresa	Sodimac
Casa	La Moneda
Empleado	Juan Pérez
Ventana (tiempo de diseño)	Ventana (tiempo de ejecución)
String	"Juan Pérez"

Definición de una Clase

```
class Circulo {
    // campos
    // métodos
    // constructores
    // main()
}
```



Campos

- Almacenamiento de información
  - Variables de instancia
  - Variables de clase (static)

Variable de Instancia

- Existe una instancia por cada objeto
- Puede ser inicializada en la declaración
- Una variable de instancia declarada final debe ser inicializada en la declaración (o en el constructor), y no puede ser modificada posteriormente

- Sintaxis: <tipo> <identificador> [= <valor inicial>];

- Ejemplo:

```
double radio = 5;
String color;
```

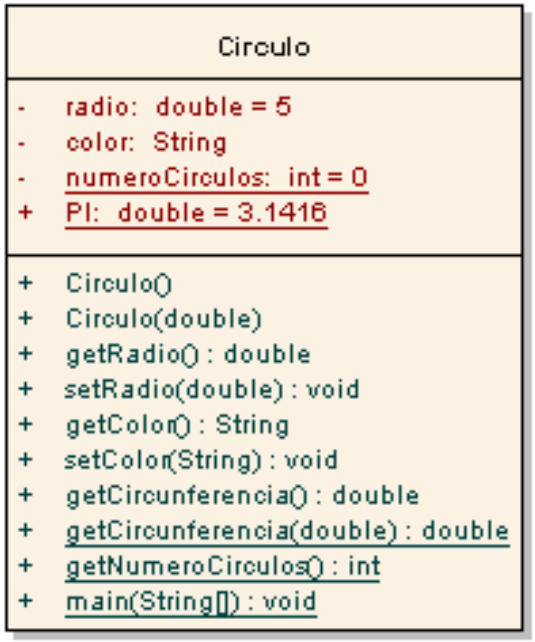
Variable de Clase (static)

- Una variable static existe una vez en memoria, independientemente del número de instancias de la clase
- Es accesible sin necesidad de instanciar la clase
- Puede ser inicializada en la declaración
- Una variable static declarada final debe ser inicializada en la declaración (o en el bloque de inicialización estática), y no puede ser modificada posteriormente
- Ejemplo:

```
static int numeroCirculos = 0;
static final double PI = 3.1416;
```

Campos en una Clase

```
class Circulo {
    // campos
    double radio = 5;
    String color;
    static int numeroCirculos = 0;
    static final double PI = 3.1416;
    // métodos
    // constructores
    // main( )
}
```



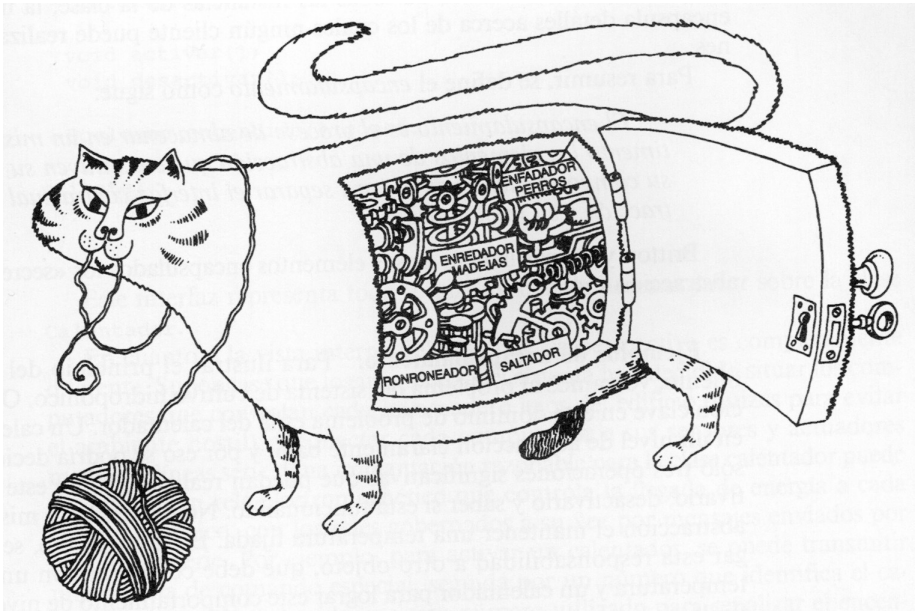
Acceso a Campos

- El acceso a miembros se realiza utilizando "dot notation"

```
Circulo c1 = new Circulo();
c1.radio = 5;
c1.color = "rojo";
Circulo.numeroCirculos++;
```

Encapsulación

- Ocultamiento de la implementación interna al usuario del objeto (cliente)



- Regla
  - Un objeto no debe acceder directamente a campos de otros objetos

Modificadores de Acceso

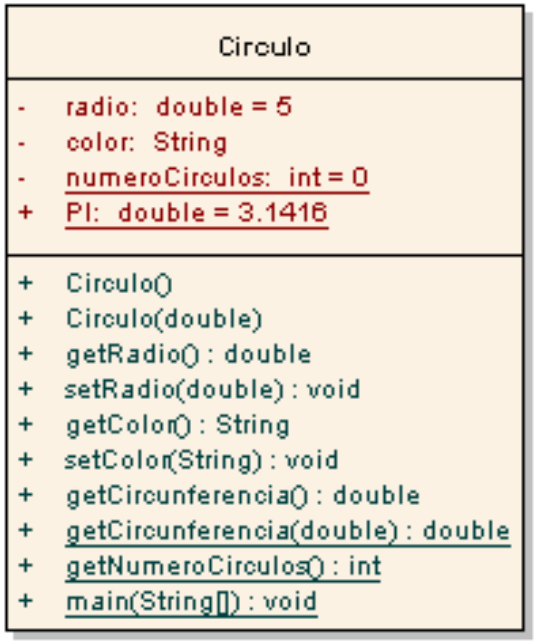
- Public: Accesible en cualquier lugar en que la clase sea accesible
- Protected: Accesible por subclases y clases del mismo package
- Package (default): Accesible por clases del mismo package
- Private: Accesible sólo al interior de la clase

Beneficios de la Encapsulación

- La encapsulación da un nivel de seguridad al código, al restringir el acceso a la implementación
- La implementación interna puede ser modificada sin afectar el código de la aplicación - sólo hay impacto en los métodos de la clase
- Al proveer un método “setter” para modificar un campo privado, éste puede realizar chequeo de errores

Definición del Acceso

```
class Circulo {
    // campos
    private double radio = 5;
    private String color;
    private static int numeroCirculos = 0;
    public static final double PI = 3.1416;
    // métodos
    // constructores
    // main( )
}
```



Métodos

- Instrucciones que operan sobre los datos de un objeto para obtener resultados
- Tienen cero o más parámetros
- Pueden retornar un valor o pueden ser declarados void para indicar que no retornan ningún valor

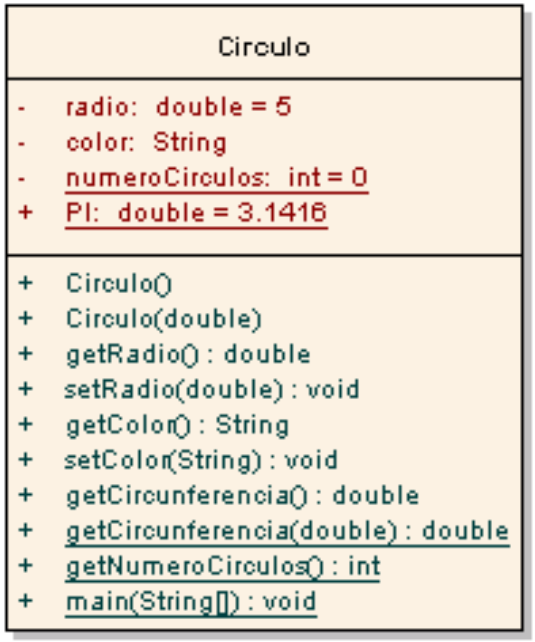
- Sintaxis

```
<tipo retorno> <nombre método> (<tipo> parámetro1, ...)
{
    // cuerpo del método
    return <valor de retorno>;
}
```

Método de Instancia

- Tiene acceso directo a las variables de instancia del objeto sobre el cual es invocado

```
double getCircunferencia()
{
  return 2 * radio * PI;
}
```

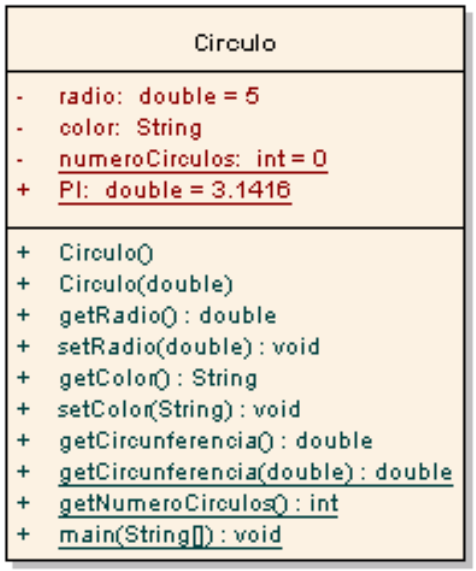


- Es invocado sobre un objeto de la clase

```
Circulo c = new Circulo();
c.setRadio(20);
double d = c.getCircunferencia();
```

Método de Clase (static)

- No actúa sobre ninguna instancia de la clase
- Puede ser utilizado sin instanciar la clase
- Sólo tiene acceso directo a variables static de la clase



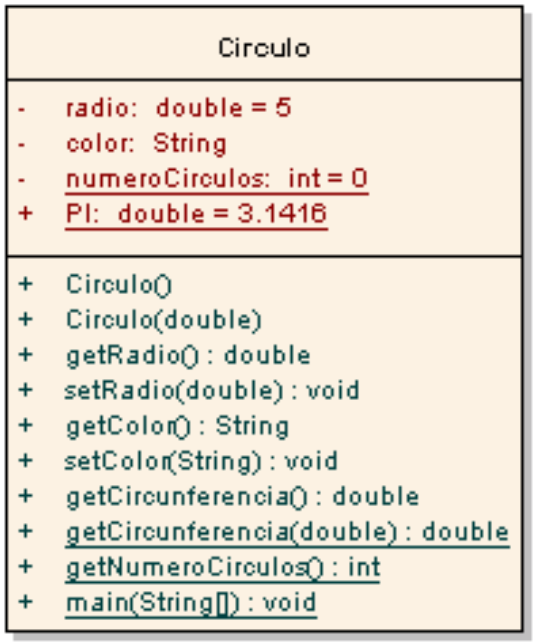

```
public static double getCircunferencia(double r) {
  return 2 * r * PI;
}
```

- Es invocado directamente sobre la clase

```
double d = Circulo.getCircunferencia(30);
```

Definición de Métodos

```
class Circulo {
    // campos
    ...
    // métodos
    public double getRadio() {...}
    public void setRadio(double radio) {...}
    public String getColor() {...}
    public void setColor(String color) {...}
    public double getCircunferencia() {...}
    public static double getCircunferencia(double radio) {...}
    public static int getNumeroCirculos() {...}
}
```



Sobrecarga de Métodos

- Sobrecarga: overloading
- Métodos de una clase pueden tener el mismo nombre pero diferentes parámetros
- Cuando se invoca un método, el compilador compara el número y tipo de los parámetros y determina qué método debe invocar
- Firma (signature) = nombre del método + lista de parámetros

Ejemplo

```
class Cuenta {
    public void depositar(double monto) {
        this.depositar(monto, "$");
    }

    public void depositar(double monto, String moneda) {
        // procesa el depósito
    }
}
```

Constructores

- Un constructor es un método especial invocado para instanciar e inicializar un objeto de una clase
  - Invocado con la sentencia new
  - Tiene el mismo nombre que la clase
  - Puede tener cero o más parámetros
  - No tiene tipo de retorno, ni siquiera void
  - Un constructor no público restringe el acceso a la creación de objetos

Constructor Default

- Es un constructor sin parámetros creado por el compilador si uno no provee ningún constructor

- Si la clase tiene algún constructor, entonces el constructor sin parámetros debe ser explícitamente creado en caso que se requiera

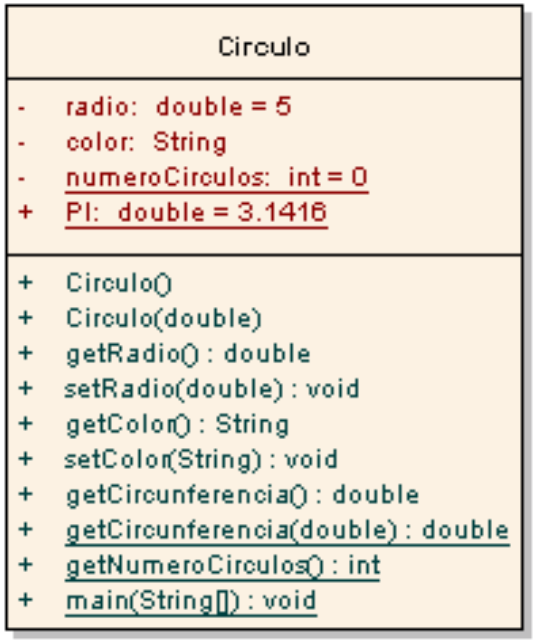
Ejemplo

```
class Circulo {
    ...

    // constructores
    public Circulo() {
    }

    public Circulo(double r) {
        radio = r;
    }

    void f() {
        Circulo c = new Circulo(30);
        ...
    }
}
```



Declaración de Variables

- La definición de una clase crea un tipo de datos
- Variables de este tipo se declaran:

```
Circulo c1;
```

- Una declaración no crea un objeto; crea una variable que contiene una referencia a un objeto, sin crear al objeto en sí

Instanciación

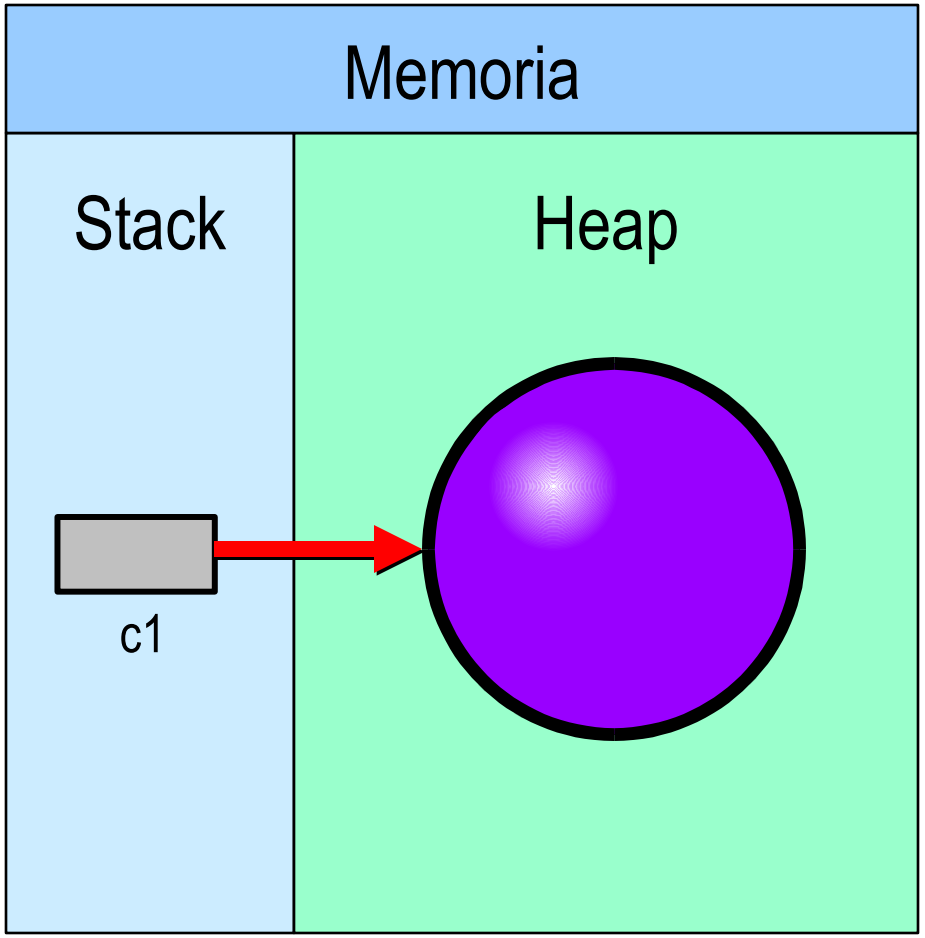
- Los objetos se crean usando el operador new

```
c1 = new Circulo();
```

- Los objetos son creados en un área de memoria conocida como el heap
- Todos los objetos son utilizados vía referencias

Instanciación

```
Circulo c1;
c1 = new Circulo();
```



Uso de Miembros de Instancia

- Para acceder a variables y métodos de instancia se utiliza la sintaxis "objeto."

```
Circulo c1 = new Circulo();
c1.radio = 5;
c1.color = "rojo";
double d = c1.getCircunferencia();
```

- Si la referencia es null, se genera una excepción NullPointerException

Uso de Miembros de Clase

- Para acceder a variables y métodos static se utiliza la sintaxis "clase."

```
Circulo c1 = new Circulo();
Circulo.numeroCirculos++;
int n = Circulo.getNumeroCirculos();
double d = Circulo.PI;
```

Uso de this

- En un método de instancia, this es una referencia al objeto sobre el cual se invocó el método

```
class Circulo {
    ...
    void grabar() {
        ...
        Database.save(this);
        LogManager.log(this);
    }
    ...
}
```

Uso de this

- La palabra this puede ser utilizada en la primera línea de un constructor para invocar a otro constructor

```
class Circulo {
    private double radio;
    private int numeroCirculos = 0;
    Circulo(double radio) {
        this.radio = radio;
        numeroCirculos++;
    }
    Circulo() {
        this(10);   // radio default: 10
    }
}
```

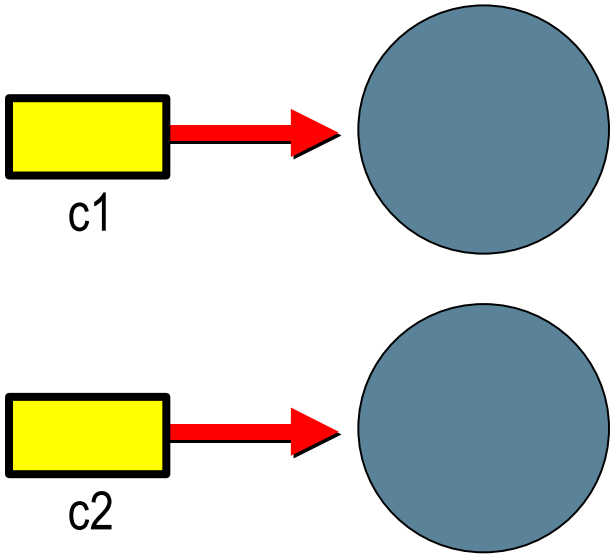
Garbage Collection

- Cuando no quedan en el programa referencias a un objeto, el espacio que él ocupa puede ser reclamado por un "garbage collector" (recolector de basura)
- Uno no elimina explícitamente objetos (no existe el delete de C++)
- Entorno seguro (no hay punteros a basura)
- Es posible invocar directamente al garbage collector para intentar forzar la recolección de basura:

```
System.gc();
```

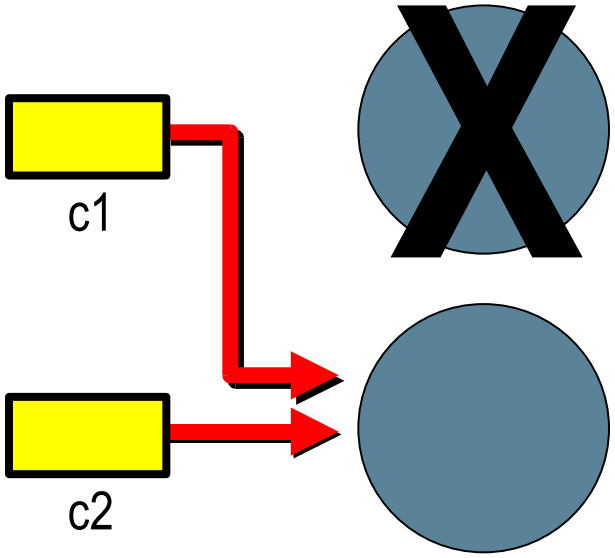
Garbage Collection

```
Circulo c1 = new Circulo();
Circulo c2 = new Circulo();
c1 = c2;
```



Garbage Collection

```
Circulo c1 = new Circulo();
Circulo c2 = new Circulo();
c1 = c2;
```

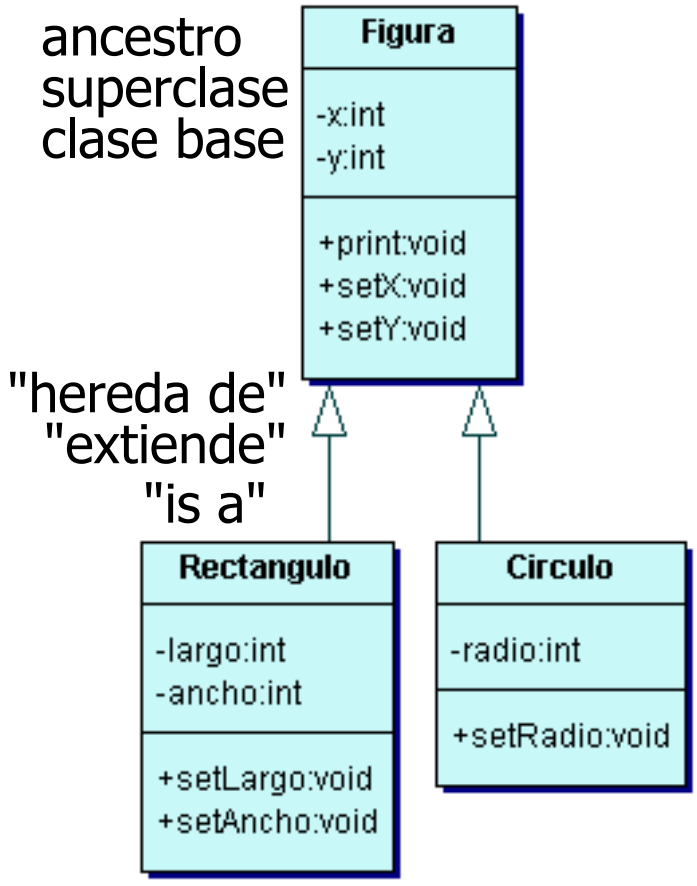


- Nota: el primer círculo no es eliminado de memoria al perderse su referencia, sino cuando la JVM decide invocar al garbage collector

Herencia

- Concepto de OOP que permite que una clase herede los campos y métodos de otra
- Todas las clases son descendientes de la clase Object
- La cláusula extends especifica el ancestro inmediato de la clase
- Una subclase o clase derivada hereda todos los campos y métodos de la superclase o clase base
- A diferencia de C++, Java no soporta herencia múltiple
- Para suplir lo anterior, Java permite que una clase implemente un conjunto de interfaces

Ejemplo




```
class Figura {
    int x, y;
    public void print() { ... }
    public void setX(int xpos) { x = xpos; }
    public void setY(int ypos) { y = ypos; }
}

class Rectangulo extends Figura {
    int largo, ancho;
    public void setLargo(int l) { largo = l; }
    public void setAncho(int a) { ancho = a; }
}


// uso:
Rectangulo r = new Rectangulo();
r.setX(10); r.setY(20);
r.setAncho(100); r.setLargo(300);
r.print();
```

Encapsulación

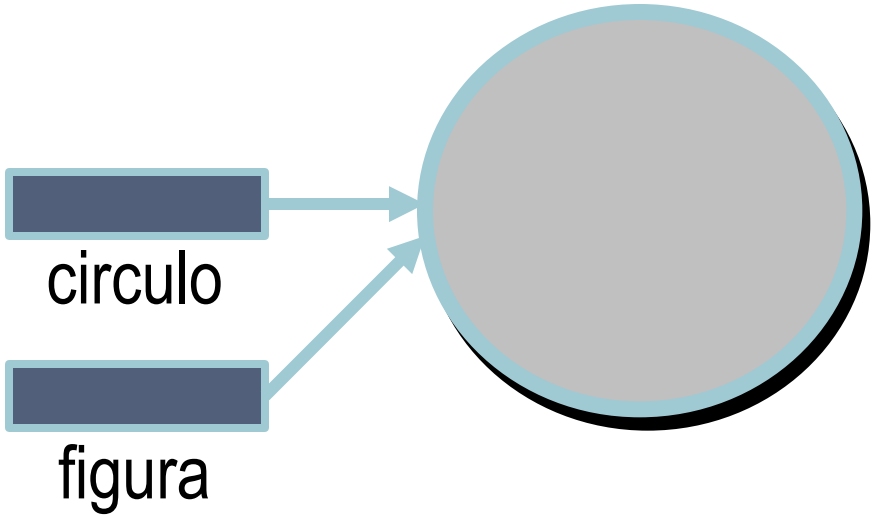
- Una clase derivada tiene acceso a los miembros public y protected de una clase base, aunque pertenezcan a paquetes diferentes

- Una clase derivada tiene acceso a los miembros package de una clase base si ambas clases pertenecen al mismo paquete

- Una clase derivada no tiene acceso a los miembros private de una clase base

Polimorfismo

```
Circulo circulo;
circulo = new Circulo();
Figura figura;
figura = circulo;
```



Compila y ejecuta bien (un círculo es una figura)
Restricción: no se puede usar figura para acceder a métodos especializados de Circulo

```
figura.getRadio();    // no compila
```

Polimorfismo

- Java permite asignar un objeto a una variable declarada con un tipo de datos ancestro

```
void metodo1(Figura f) {
    f.print();
    ...
}

void metodo2() {
    metodo1(new Circulo());
}
```

Redefinición de Métodos

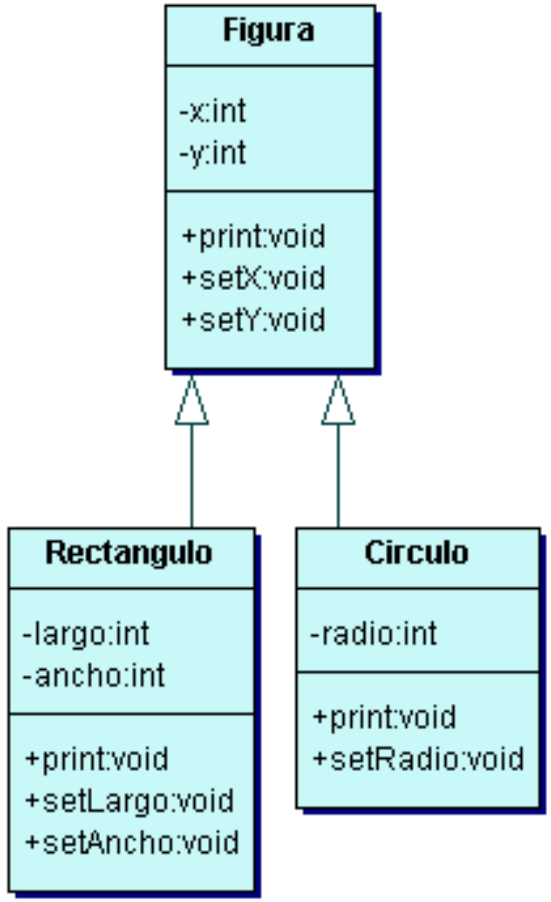
- Redefinición: overriding

- Reemplazar la implementación de un método existente en la clase base
  - La firma y el tipo de retorno deben ser idénticos a los de la clase base (de lo contrario se produce sobrecarga)
  - El nivel de acceso puede ser el mismo o mayor (un método protegido no puede ser redefinido como privado)

Ejemplo

```
public class Figura {
    public void print() {
        ...
    }
}

public class Circulo extends Figura {
    public void print() {
        ...
    }
}
```



La Palabra Clave super

- La palabra clave super puede ser utilizada para invocar explícitamente la implementación de un método de la clase base
- Hace referencia al ancestro inmediato
- Disponible en métodos de instancia

```
public class Circulo extends Figura {
  public void save() {
    super.save();
    ...
  }
}
```

Dynamic Binding

- Al invocar un método de instancia, el tipo real (dinámico) del objeto — no el tipo de la referencia (estático) — es utilizado para determinar qué versión del método invocar

```
void miMetodo(Figura f) {
    ...
    f.print(); // invoca a Circulo.print() si f es
               // una referencia a un Circulo
    ...
}
```

Compatibilidad de Tipos

- Java es fuertemente tipeado, exige compatibilidad de tipos en tiempo de compilación:
  - Permite asignar un objeto a una variable de un tipo ancestro
  - Requiere un cast explícito para asignar un objeto a una variable de un tipo descendiente

    ```
    Circulo c = (Circulo) figura;
    ```
  - Si el cast falla en ejecución, la máquina virtual lanza un ClassCastException
  - El compilador no permite realizar una conversión de un objeto a un tipo que no es ancestro ni descendiente

Identificación de Tipo

- El método getClass() de la clase Object retorna un objeto de tipo Class correspondiente a la clase real a la que pertenece el objeto

- El operador instanceof indica si un objeto es de una clase determinada o de alguna clase descendiente

```
if (figura instanceof Circulo) {
    Circulo circulo = (Circulo) figura;
    // uso de circulo
}
```

Constructor en Subclases

- El constructor de una subclase **debe** invocar algún constructor de la clase base:
  - Explícitamente: usando **super()** en la primera línea
  - Implícitamente: si no se invoca el constructor de la clase base explícitamente, se invoca el constructor default

- Ejemplo

```
class Rectangulo extends Figura {
    int largo, ancho;
    Rectangulo(int x, int y, int l, int a) {
        super(x, y);
        largo = l;
        ancho = a;
    }
}
```

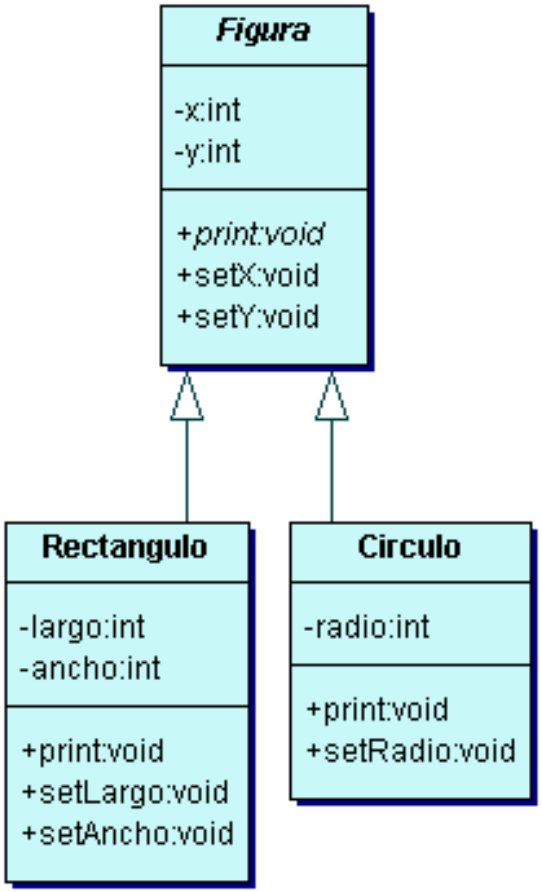
Clases Abstractas

- Una clase abstracta no puede ser instanciada, lo cual es asegurado por el compilador
- Puede contener métodos abstractos, a ser implementados en subclases
- Puede contener métodos concretos
- Una subclase de una clase abstracta debe:
  - implementar todos los métodos abstractos heredados, o bien
  - ser a su vez declarada abstracta

Ejemplo

```
public abstract class Figura {
    public abstract void print();
}

public class Rectangulo extends Figura {
    public void print() {
    // implementa print para Rectangulo
  }
}
```



Diseño Tradicional

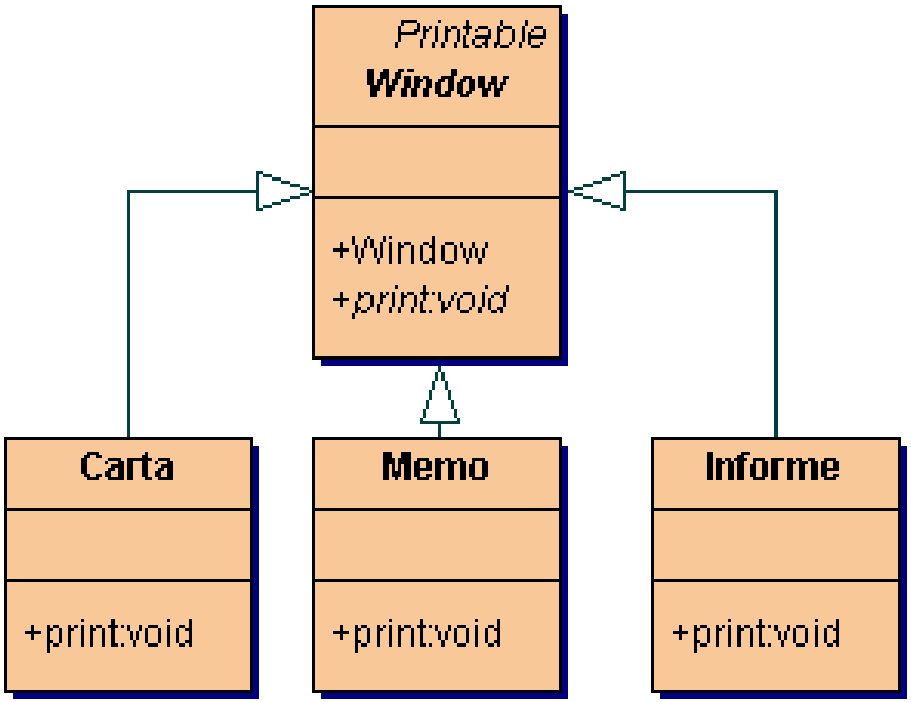
- Supongamos que se desea construir una aplicación de múltiples ventanas (cartas, memos, etc.), con un menú global que permita imprimir la ventana activa

```
// En un lenguaje tradicional: C
void print()
{
    window* w = getCurrentWindow();
    switch (w->type) {
        case CARTA: printCarta(w); break;
        case MEMO:  printMemo(w);  break;
        ...
    }
}
```

Diseño con Polimorfismo

```
// La capa genérica
abstract class Window {
    public abstract void print();
}
class Carta extends Window {
    public void print() {...}
}
class Memo extends Window {
   ...

// La aplicación
class MiAplicacion {
    void print() {
        GUI.getCurrentWindow().print();
    }
}
```

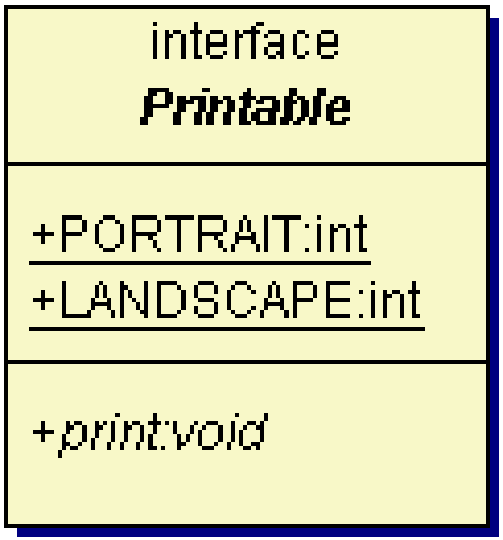


Interfaces

- Java no soporta herencia múltiple, pero sí permite que una clase implemente múltiples interfaces
- Una interfaz (interface) es una colección de métodos abstractos y constantes
- No puede ser instanciada
- Define un tipo de datos que se puede utilizar en la declaración de variables
- Java soporta herencia múltiple de interfaces

Ejemplo

```
interface Printable {
    int PORTRAIT = 0;
    int LANDSCAPE = 1;
    void print(int orientacion);
}
```



- Campos son automáticamente public final static
- Métodos son automáticamente public abstract

Implementando una Interfaz

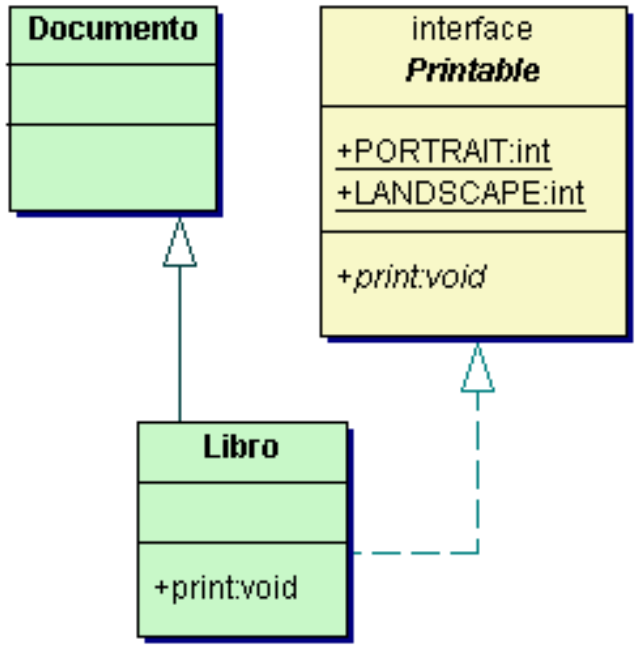
- Una clase puede implementar un número ilimitado de interfaces
- Una clase que implementa interfaces debe:
  - Implementar todos los métodos definidos en las interfaces, o bien
  - Ser declarada abstract

Implementando una Interfaz

```
class Libro extends Documento implements Printable
{
    public void print(int orientacion) {
        // implementación
    }
}

class Empleado implements Printable {
    public void print() { ... }
}

class Rectangulo implements Printable {
    public void print() { ... }
}
```



Uso de Interfaces

- Una definición de una interfaz crea un tipo de datos
- No es posible instanciar este tipo
- Sí es posible declarar variables de este tipo
- Es posible asignar a estas variables objetos de clases que implementen la interfaz
- Ejemplo

```
Printable printable = new Libro();
```

Uso de Interfaces

- Capa genérica

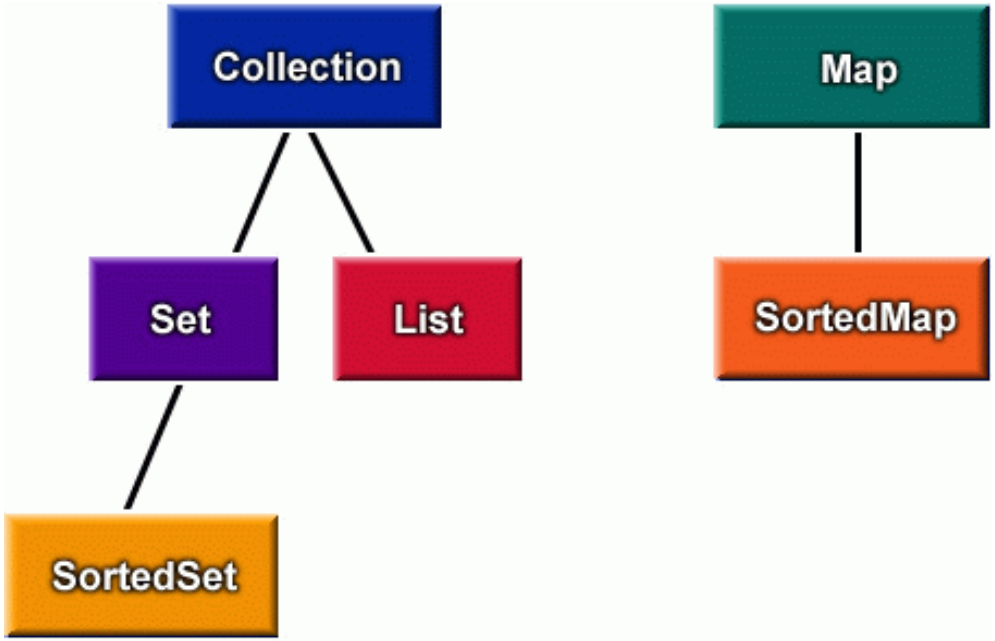
```
class ColaImpresion {
    static void creaJob(Printable p) {
        …
        p.print();
    }
}
```

- Capa cliente

```
ColaImpresion.creaJob(new Empleado(…));
ColaImpresion.creaJob(new Rectangulo(…));
Libro libro = new Libro();
…
ColaImpresion.creaJob(libro);
```

Framework de Colecciones

- Java provee en el package java.util un framework de colecciones que contiene dos jerarquías de clases:
  - Collection: grupo de *elementos*, permite agregar elementos a la colección y luego recorrerlos
  - Map: objeto que mapea *llaves* con *valores*, permite asociar un objeto a una llave, y posteriormente obtener el objeto asociado a una llave



Principales Implementaciones

- Listas:
  - ArrayList: eficiente en accesos
  - LinkedList: eficiente en inserciones/eliminaciones
- Set:
  - HashSet: recorridos desordenados
  - TreeSet: recorridos ordenados
- Map:
  - HashMap: recorridos desordenados
  - TreeMap: recorridos ordenados según orden de llaves

		Implementations			
		Hash Table	Resizable Array	Balanced Tree	Linked List
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	

Colecciones

- Collection
  - Grupo de objetos, conocidos como sus elementos
  - Puede permitir o no elementos duplicados
  - Puede ser ordenada o no
  - No tiene implementaciones directas
- Set
  - Colección no ordenada sin elementos duplicados
  - Implementaciones: AbstractSet, HashSet, LinkedHashSet, TreeSet
- List
  - Colección ordenada, puede contener elementos duplicados
  - Incluye operaciones para acceso posicional
  - Implementaciones: AbstractList, ArrayList, LinkedList, Vector
- Iterator
  - Permite recorrer una colección

Las Interfaces Collection e Iterator

```
public interface Collection<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);                 // opcional
    boolean remove(Object element);         // opcional
    void clear();                           // opcional
    Iterator<E> iterator();
    Object[] toArray();
    ...
}

public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();                          // opcional
}
```

Ejemplos

- Agregando elementos a una colección

```
Collection collection = new ArrayList();
collection.add("Hola, qué tal!");
collection.add("Chao!");
```

- Con Java 5

```
Collection<String> collection = new ArrayList<String>();
collection.add("Hola, qué tal!");
collection.add("Chao!");
```

- Recorriendo una colección

```
Iterator iterator = collection.iterator();
while (iterator.hasNext()) {
    String s = (String)iterator.next();
    ...
}
```

- Con Java 5

```
for (String s : collection) { ... }
```

La Interfaz Map

- Un Map mapea *llaves* a *valores*
- Un Map no permite llaves duplicadas
- Dos implementaciones principales
  - HashMap: almacena los elementos en un hash table, y tiene el mejor desempeño
  - TreeMap: almacena los elementos ordenados ascendentemente por sus llaves; el ordenamiento puede estar basado en el "orden natural", o en un Comparator particular, al igual que en TreeSet

La Interfaz Map

```
public interface Map<K,V> {
    V put(K key, V value);
    V get(K key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
    void clear();
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();
    ...

    public static interface Entry<K,V> {
        K getKey();
        V getValue();
        ...
    }
}
```

La Interfaz Map

- Agregando un objeto a un Map

```
Map<String,Producto> map =
        new HashMap<String,Producto>();
map.put("A100", new Producto("A100", "Camisa", ...));
```

- Recuperando un objeto de un Map

```
Producto producto = map.get("A100");
```

- Iterando sobre las llaves:

```
for (String key : map.keySet()) {...}
```

- Iterando sobre los valores:

```
for (Producto producto : map.values()) {...}
```

- Iterando sobre los pares {llave,valor}:

```
for (Map.Entry entry : map.entrySet()) {
    String key = entry.getKey();
    Producto producto = entry.getValue();
}
```

Algoritmos

- Java provee un conjunto de algoritmos polimórficos en la clase Collections: binarySearch(), copy(), fill(), max(), min(), replaceAll(), reverse(), rotate(), shuffle(), sort(), swap()
- La mayoría opera sobre objetos de tipo List, pero algunos (min y max) operan sobre objetos Collection

Ordenamiento

- Dos formas de ordenamiento:
  - La interfaz java.lang.Comparable provee el ordenamiento "natural" para las clases que la implementan (Character, Date, Double, File, Float, Integer, Long, Short, String, URI, entre otras)

```
interface Comparable {
    int compareTo(Object o);
}
```

  - La interfaz java.util.Comparator entrega al programador un control completo sobre el ordenamiento

```
interface Comparator {
    int compare(Object o1, Object o2);
}
```

Resumen

- Una clase es una plantilla a partir de la cual se instancian objetos
- Los objetos contienen información (en variables de instancia y de clase) y comportamiento (en métodos de instancia y de clase), y se manejan a través de referencias
- Los modificadores de acceso controlan quién tiene acceso a los campos y métodos de una clase
- Usando herencia, una clase adquiere las propiedades y los métodos de otra
- Es posible asignar a una variable objetos de clases que implementen el tipo de la variable
- Si se invoca un método que ha sido redefinido, el método invocado es el del tipo del objeto en ejecución (no el tipo de la variable)

Resumen

- Una clase abstracta puede contener métodos abstractos, y no puede ser instanciada
- Una interfaz es una colección de métodos abstractos y constantes, y no puede ser instanciada
- Una clase puede extender a una clase, e implementar un número ilimitado de interfaces
- Java provee un framework de colecciones que incluye las interfaces Collection, Set, List, Iterator, Map
- Es posible agregar elementos a un Collection, y luego recorrerlos
- Un Map permite asociar valores a llaves, y luego obtener el valor asociado a una llave