ST0244 Lenguajes de Programacion
Programación orientada a objetos

Andrés Sicard Ramírez

Universidad EAFIT

Semestre 2024-2

Preliminares

Convenciones

- La numeración (capítulos, teoremas, figuras, páginas, etc) en estas diapositivas corresponde a la numeración del texto guía [Louden y Lambert 2011].
- Los ejemplos que incluyen código fuente están en el repositorio del curso.

Introducción: Línea de tiempo

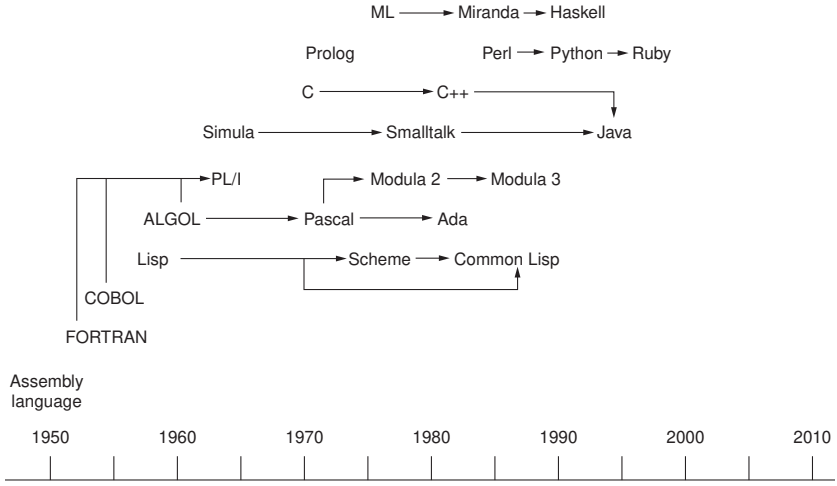


Figura 1.1

Introducción

Inicio

«One of the central goals of this project [the Simula project] was to incorporate into the language the notion of an object, which, similar to a real-world object, is an entity with certain properties that control its ability to react to events in predefined ways. According to this way of looking at programming, a program consists of a set of objects, which can vary dynamically, and which execute by acting and reacting to each other, in much the same way that a real-world process proceeds by the interaction of real-world objects.» (pág. 142)

Introducción

Características

La programación orientada a objetos satisface tres importantes necesidades en el diseño de software (pág. 143):

(i) *«the need to reuse software components as much as possible»*,

(ii) *«the need to modify program behavior with minimal changes to existing code»* and

(iii) *«the need to maintain the independence of different components»*.

Introducción

Concepto		Cuenta	%
Herencia	(*Inheritance*)	71	81 %
Objeto	(*Object*)	69	78 %
Clase	(*Class*)	62	71 %
Encapsulación	(*Encapsulation*)	55	63 %
Método	(*Method*)	50	57 %
Paso de mensajes	(*Message passing*)	49	56 %
Polimorfismo	(*Polymorphism*)	47	53 %

«*The quarks of object-oriented development* » [Armstrong 2006].

Introducción

Definición de los *quarks* [Armstrong 2006]

(i) «***Inheritance:*** *A mechanism that allows the data and behavior of one class to be included in or used as the basis for another class.*»

(ii) «***Object:*** *Individual, identifiable item, either real or abstract, which contains data about itself and descriptions of its manipulations of the data.*»

(iii) «***Class:*** *A description of the organization and actions shared by one or more similar objects.*»

(iv) «***Encapsulation:*** *A technique for designing classes and objects that restricts access to the data and behavior by defining a limited set of messages that an object of that class can receive.*»

(continua en la próxima diapositiva)

Introducción

Quark's definitions (continuation)

(v) «***Method:*** *A way to access, set or manipulate object's information.*»

(vi) «***Message passing:*** *The process by which an object sends data to another object or asks the other object to invoke a method.*»

(vii) «***Polymorphism*** *is defined as: the ability of different classes to respond to the same message and each implement the method appropriately.*»

Introducción

Lenguajes de programación

Algunos lenguajes orientados a objetos son C++, Java, Python, Ruby, Scala y Smalltalk.

Introducción a Smalltalk

Características importantes

- Smalltalk es un lenguaje orientado a objetos puro. Cada entidad del lenguaje, incluyendo los números, es un objeto.
- En Smalltalk los objetos interactúan a través del paso de mensajes.
- La herencia de las clases es simple (solo se puede heredar de una clase).
- El chequeo de tipos en Smalltalk es dinámico.
- Smalltalk tiene recolector de basura (*garbage collector*) que elimina de la memoria los objetos que ya no se utilizan.

Introducción a Smalltalk

Orígenes

«Los orígenes de Smalltalk se encuentran en las investigaciones realizadas por Alan Kay, Dan Ingalls, Ted Kaehler, Adele Goldberg y otros durante los años setenta en el Palo Alto Research Center de Xerox (conocido como Xerox PARC), para la creación de un sistema informático orientado a la educación. El objetivo era crear un sistema que permitiese expandir la creatividad de sus usuarios, proporcionando un entorno para la experimentación, creación e investigación.» (Wikipedia)

Introducción a Smalltalk

Usando Smalltalk

(i) La implementación de Smalltalk usada en el curso es GNU Smalltalk la cual puede ser instalada con el siguiente comando:

```
$ sudo apt install gnu-smalltalk
```

Introducción a Smalltalk

Usando Smalltalk

(i) La implementación de Smalltalk usada en el curso es GNU Smalltalk la cual puede ser instalada con el siguiente comando:

```
$ sudo apt install gnu-smalltalk
```

(ii) Para ejecutar el interpretador de Smalltalk use el comando

```
$ gst
GNU Smalltalk ready

st>
```

Clases, objetos, mensajes y control

- Cada objeto tiene un conjunto de **propiedades** (*properties*) y **comportamientos** (*behaviors*).
- Los objetos interactúan a través del **paso de mensajes** (*message passing*).
- El objeto que envía el mensaje es el **emisor** (*sender*) y el objeto que recibe el mensaje es el **receptor** (*receiver*).
- Un **mensaje** es una solicitud por un servicio.
- El receptor realiza el servicio solicitado invocando un **método** (el cual puede enviar otros mensajes a otros objetos para llevar a cabo la tarea).
- Un método puede retornar datos al emisor del mensaje.
- El emisor del mensaje puede suministrar datos vía argumentos.
- La sintaxis del paso de mensajes es: objeto que recibe el mensaje (receptor), el nombre del mensaje (método llamado **selector**) y los argumentos.

Ejemplo

Creamos un nuevo objeto por enviar el mensaje `new` a la clase *built-in* `Set`.

```
st> Set new
Set ()
```

Clases, objetos, mensajes y control

Ejemplo

Creamos un nuevo objeto por enviar el mensaje `new` a la clase *built-in* `Set`.

```
st> Set new
Set ()
```

Ejemplo

Preguntamos por el número de elementos de un conjunto nuevo enviando el mensaje `size` a este objeto.

```
st> Set new size
0
```

Clases, objetos, mensajes y control

- Un **mensaje a un clase (*class message*)** es un mensaje enviado a una clase.
- Un **mensaje a una instancia (*instance message*)** es un mensaje enviado a una instancia de una clase.
- Un **mensaje unario (*unary message*)** es un mensaje sin argumentos.
- Un ***keyword message*** es un mensaje con argumentos.
- Un mensaje unario tiene mayor precedencia que un *keyword message*.

Clases, objetos, mensajes y control

Ejemplo (mensaje unario a una clase)

El mensaje `new` es un mensaje unario enviado a la clase *built-in* `Set`.

```
st> Set new
Set ()
```

Clases, objetos, mensajes y control

Ejemplo (mensaje unario a una clase)

El mensaje `new` es un mensaje unario enviado a la clase *built-in* `Set`.

```
st> Set new
Set ()
```

Ejemplo (mensaje unario a una instancia)

El mensaje `size` es un mensaje unario enviado a una instancia de la clase `Set`.

```
st> Set new size
0
```

Ejemplo (*keyword message*)

Preguntamos si un conjunto contiene un elemento enviándole el *keyword message* `include:` y el elemento por el cual queremos preguntar.

```
st> Set new includes: 'Hello'
false
```

Clases, objetos, mensajes y control

- Un **mutador (*mutator*)** es un mensaje que cambia el estado del objeto receptor.

Ejemplo (mutador)

Adicionamos un nuevo elemento a un conjunto enviándole el mutador `add:` junto con el elemento que queremos adicionar.

```
st> Set new add: 'Ken'
Ken
```

Ejemplo (mutador)

Adicionamos un nuevo elemento a un conjunto enviándole el mutador `add:` junto con el elemento que queremos adicionar.

```
st> Set new add: 'Ken'
Ken
```

Ejemplo (mutador)

Creamos un arreglo de 5 elementos y almacenamos el valor 42 en la primera posición del arreglo usando el mutador de dos argumentos `at:put:`.

```
st> (Array new: 5) at: 1 put: 42
42
```

Clases, objetos, mensajes y control

- Expresiones aritméticas y comparaciones son escritas usando **mensajes binarios (*binary messages*)**.
- Los mensaje binarios tienen menor precedencia que los *keyword messages*.

Clases, objetos, mensajes y control

Ejemplo (mensajes binarios)

```
st> 3 + 4
7

st> 3 < 4
true
```

Variables e instrucciones

Ejemplo

Mirar el archivo `oop/instructions.st`.

```
$ cd oop
$ gst instructions.st
(10 20 30 40 50 )
(10 20 30 40 50 )
```

Semántica por valor y semántica por referencia

Descripción

(i) En una **semántica por valor (*value semantics*)** una variable es un objeto.

(ii) En una **semántica por referencia (*reference semantics*)** una variable se refiere a un objeto.

Semántica por valor y semántica por referencia

Descripción

(i) En una **semántica por valor (*value semantics*)** una variable es un objeto.

(ii) En una **semántica por referencia (*reference semantics*)** una variable se refiere a un objeto.

Observación

«The sharing of objects in reference semantics enhances the efficiency of object-oriented programs, at the cost of possible safety problems because of aliasing.» (pág. 148).

Semántica por valor y semántica por referencia

Ejemplo

(i) En Lisp y Smalltalk las variables usan una semántica por referencia.

(ii) En Java las variables cuyo tipo es un objeto usan una semántica por referencia y las variables cuyo tipo es un *primitive data types* usan una semántica por valor.

(iii) En Ada y C++ las variables usan una semántica por valor.

Semántica por valor y semántica por referencia

Ejemplo (pág. 147)

En Smalltalk el operador = es para igualdad y el operador == es para la identidad entre objetos.

```
| array alias clone |
array := #(0 0).     "Create an array containing two integers"
alias := array.      "Establish a second reference to the same object"
clone := #(0 0).     "Create a copy of the first array object"
array == alias.      "true, they refer to the exact same object"
array = alias.       "true, because they are =="
array = clone.       "true, because they refer to distinct objects with"
                     "the same properties"
array == clone       "false, because they refer to distinct objects"
clone at: 1 put: 1. "The second array object now contains 1 and 0"
array = clone        "false, because the two array objects no longer"
                     "have the same properties"
```

Ciclo `to:do:`

Ejemplo

```
1 to: 20 do: [ :x | x printNl ]
```

Ciclo `to:do:`

Ejemplo

```
1 to: 20 do: [ :x | x printNl ]
```

Ejemplo (pág. 147)

```
1 | array |
2 array := Array new: 100.
3 1 to: array size do: [ :i | array at: i put: i * 10 ]
```

Condicional `ifTrue:ifFalse:`

Ejemplo (pág. 149)

Usando el mensaje `ifTrue:ifFalse:` para establecer los valores de un arreglo en posiciones impares en 1 y los valores en posiciones pares en 2.

```
| array |
array := Array new: 100.
1 to: array size do: [ :i |
  i odd
    ifTrue: [array at: i put: 1]
    ifFalse: [array at: i put: 2]]
```

Jerarquía de clases, herencia y polimorfismo

Definición

*«**Inheritance** is: a mechanism that allows the data and behavior of one class to be included in or used as the basis for another class.» (Armstrong 2006, pág. 124)*

Jerarquía de clases, herencia y polimorfismo

Definición

*«**Inheritance** is: a mechanism that allows the data and behavior of one class to be included in or used as the basis for another class.» (Armstrong 2006, pág. 124)*

*«**Inheritance** is a relationship among classes wherein one class shares the structure and/or behavior defined in one (**single inheritance**) or more (**multiple inheritance**) other classes. We call the class from which another class inherits its **superclass**... Similarly, we call a class that inherits from one or more classes a **subclass**.» (Booch, Maksimchuk, Engle, Young, Conallen y Houston 2007, pág. 100)*

Jerarquía de clases, herencia y polimorfismo

Definición

*«**Inheritance** is: a mechanism that allows the data and behavior of one class to be included in or used as the basis for another class.» (Armstrong 2006, pág. 124)*

*«**Inheritance** is a relationship among classes wherein one class shares the structure and/or behavior defined in one (**single inheritance**) or more (**multiple inheritance**) other classes. We call the class from which another class inherits its **superclass**... Similarly, we call a class that inherits from one or more classes a **subclass**.» (Booch, Maksimchuk, Engle, Young, Conallen y Houston 2007, pág. 100)*

*«**Inheritance**, thus, supports the reuse of structure and behavior.» (pág. 150)*

Jerarquía de clases, herencia y polimorfismo

Definición

*«**Polymorphism** is defined as: the ability of different classes to respond to the same message and each implement the method appropriately.» (Armstrong 2006, pág. 126)*

Jerarquía de clases, herencia y polimorfismo

Definición

*«**Polymorphism** is defined as: the ability of different classes to respond to the same message and each implement the method appropriately.» (Armstrong 2006, pág. 126)*

*«**Polymorphism** is a concept in type theory wherein a name may denote instances of many different classes as long as they are related by some common superclass... With polymorphism, an operation can be implemented differently by the classes in the hierarchy. In this manner, a subclass can extend the capabilities of its superclass or override the parent's operation.» (Booch, Maksimchuk, Engle, Young, Conallen y Houston 2007, pág. 102)*

Jerarquía de clases, herencia y polimorfismo

Definición

*«**Polymorphism** is defined as: the ability of different classes to respond to the same message and each implement the method appropriately.» (Armstrong 2006, pág. 126)*

*«**Polymorphism** is a concept in type theory wherein a name may denote instances of many different classes as long as they are related by some common superclass. . . With polymorphism, an operation can be implemented differently by the classes in the hierarchy. In this manner, a subclass can extend the capabilities of its superclass or override the parent's operation.» (Booch, Maksimchuk, Engle, Young, Conallen y Houston 2007, pág. 102)*

*«**Polymorphism**, or the use of the same names for messages requesting similar services from different classes, is also a form of code reuse.» (pág. 150)*

Jerarquía de clases, herencia y polimorfismo

Observación

El polimorfismo de la programación orientada a objetos es llamado **run-time polymorphism** o **subtype polymorphism**.

Jerarquía de clases, herencia y polimorfismo

Descripción

- En Smalltalk las clases *built-in* son organizadas en una jerarquía en forma de árbol, con una clase llamada `Object` en la raíz.
- Cada clase abajo de la clase `Object` hereda las propiedades y los comportamientos definidos en cualquier clase que sea su ancestro en la jerarquía.
- Las clases en la jerarquía usualmente descienden de lo más general o lo más específico.
- Cada clase nueva adiciona/modifica el comportamiento y/o las propiedades de las clases de las cuales hereda éstos.

Jerarquía de clases, herencia y polimorfismo

Ejemplo (la clase *Magnitude* en Smalltalk/V)

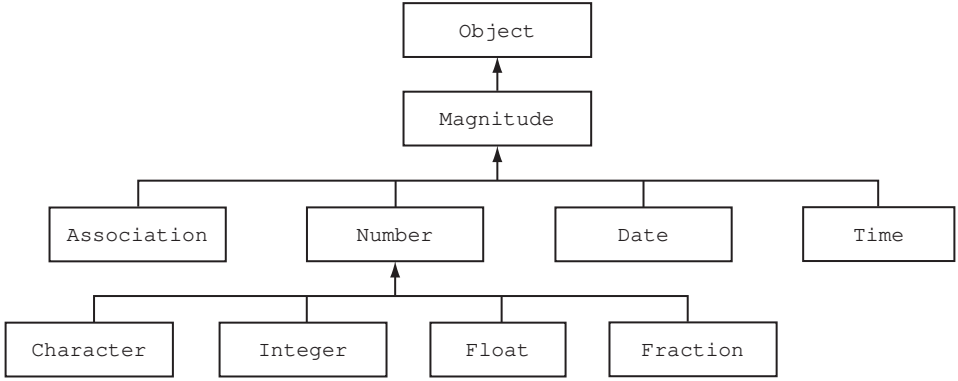


Figura 5.1

Jerarquía de clases, herencia y polimorfismo

Ejemplo (números complejos)

El conjunto de los números complejos, denotado $\mathbb{C}$, está definido por:

$$\mathbb{C} = \{\, (a, b) \mid a, b \in \mathbb{R} \,\}.$$

Sean (a, b) y (c, d) dos números complejos, entonces

$$\begin{aligned} (a, b) &= (c, d) \text{ sii } a = c \text{ y } b = d, \\ (a, b) + (c, d) &:= (a + c, b + d), \\ (a, b) \cdot (c, d) &:= (ac - bd, ad + bc). \end{aligned}$$

Un número complejo (a, b) se puede escribir en la forma rectangular $a + bi$, donde $i := (0, 1)$ es la unidad imaginaria, a es la parte real y b es la parte imaginaria.

(continua en la próxima diapositiva)

Jerarquía de clases, herencia y polimorfismo

Ejemplo (números complejos (continuación))

Mirar el archivo `oop/complex-numbers/smalltalk/Complex.st`.

```
$ cd oop/complex-numbers/smalltalk/
$ gst
st> FileStream fileIn: 'Complex.st'
```

Jerarquía de clases, herencia y polimorfismo

Ejemplo (sistema de cuentas bancarias simplificado)

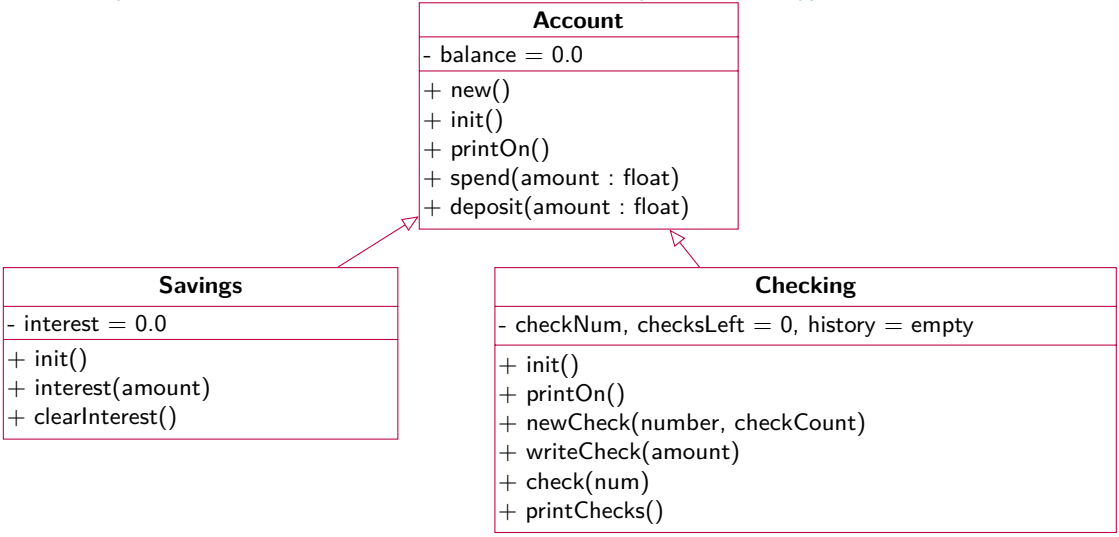
El tutorial de GNU Smalltalk* describe la implementación de un sistema de cuentas bancarias simplificado empleando tres clases: *Account* (Cuenta), *Savings* (Cuenta de Ahorros) y *Cheking* (Cuenta Corriente).

(continua en la próxima diapositiva)

*En https://www.gnu.org/software/smalltalk/manual/html_node/Tutorial.html.

Jerarquía de clases, herencia y polimorfismo

Ejemplo (sistema de cuentas bancarias simplificado (continuación))



Jerarquía de clases, herencia y polimorfismo

Ejemplo (sistema de cuentas bancarias simplificado (continuación))

Mirar el archivo `oop/toy-accounting-system/smalltalk/TAS.st`.

```
$ cd oop/toy-accounting-system/smalltalk/
$ gst
st> FileStream fileIn: 'TAS.st'
```

Introducción a Java

Características importantes

- Eslogan: *write once, run anywhere*
- Java tiene ocho *primitive data types*: Cuatro tipos de datos para números enteros (`byte`, `short`, `int`, `long`), dos tipos de datos para números en punto flotante (`float` y `double`), `char` y `boolean`.
- Java es un lenguaje orientado a objetos puro con una excepción. Los *primitive data types* no son objetos por razones de eficiencia.
- Java es un lenguaje portable (independiente de la plataforma). Sus programas son compilados a instrucciones *bytecode* las cuales son ejecutadas por una máquina virtual.

(continua en la próxima diapositiva)

Introducción a Java

Características importantes

- Java proporciona un entorno seguro para el desarrollo y ejecución de programas. No hay apuntadores en Java y los programas son ejecutados en una máquina virtual.
- Java tiene recolector de basura (*garbage collector*) el cual es ejecutado por la máquina virtual.
- En Java los parámetros son evaluados antes de ser pasados a los métodos.
- El chequeo de tipos en Java es estático.
- Java provee un extenso conjunto de bibliotecas.

Comentario del texto guía

«Java's portability, small code footprint, conventional syntax, and support for compile-time type checking gave it an immediate advantage over Smalltalk. Nevertheless, it adopted many of the features of Smalltalk, including runtime garbage collection and the use of references instead of explicit pointers, which make Java a safer language than C or C++.» (pág. 163)

Introducción a Java

Comentario del texto guía

> *«Java's portability, small code footprint, conventional syntax, and support for compile-time type checking gave it an immediate advantage over Smalltalk. Nevertheless, it adopted many of the features of Smalltalk, including runtime garbage collection and the use of references instead of explicit pointers, which make Java a safer language than C or C++.» (pág. 163)*

Observación

Safe C++ es una reciente propuesta para extender C++ con un subconjunto rigurosamente seguro.

Introducción a Java

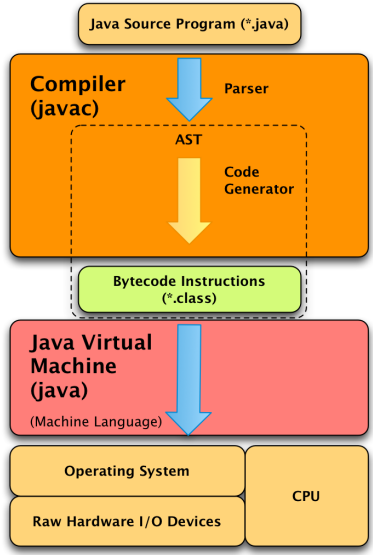
Orígenes

El lenguaje de programación Java fue desarrollado originalmente por James Gosling, de Sun Microsystems. El lenguaje apareció por primera vez en mayo 23 de 1995 como un componente fundamental de la plataforma Java de Sun.

Herramientas

- Compilador: Convierte un programa fuente en Java a *bytecode*.
- *Java Virtual Machine (JVM)*: Ejecuta el *bytecode*.

(Figura tomada de [Lee 2017, Fig. 4.4].)



Usando Java

Compilador y máquina virtual

La implementación de Java usada en el curso es la de OpenJDK la cual puede ser instalada con la siguiente instrucción:

```
$ sudo apt install openjdk-21-jdk
```

Usando Java

Compilador y máquina virtual

(i) Para determinar la versión instalada del compilador use la siguiente instrucción:

```
$ javac --version
javac 21.0.4
```

(ii) Para determinar la versión instalada de la máquina virtual use la siguiente instrucción:

```
$ java --version
openjdk 21.0.4 2024-07-16
```

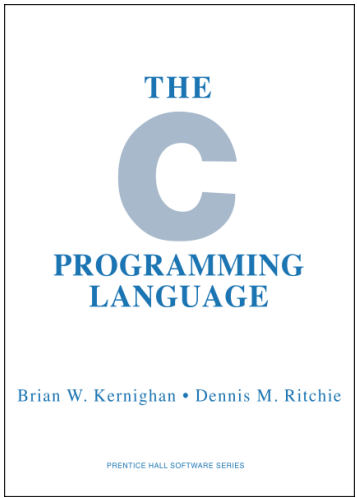
Usando Java

Compilador y máquina virtual

Para compilar y ejecutar el programa `hw/HelloWorld.java` en el repositorio use las siguiente instrucciones:

```
$ cd st0244-pl/hw
$ javac HelloWorld.java
$ java HelloWorld
Hello World! (Java)
```

Acerca del ejemplo «hello, world»



«*The first program to write is the same for all languages: Print the words* `hello, world`.» [1978, § 1.1]

Opciones del compilador

Compilación del programa `oop/Warning.java` con la opción `-Xlint`:

```
$ cd st0244-pl/oop
$ javac -Xlint Warning.java
Warning.java:4: warning: [cast] redundant cast to String
      String s = (String)"hello, word!";
                 ^
1 warning
```

Opciones del compilador

Compilación del programa `oop/Warning.java` con las opciones `-Xlint` y `-Werror`:

```
$ cd st0244-pl/oop
$ javac -Xlint -Werror Warning.java
Warning.java:4: warning: [cast] redundant cast to String
      String s = (String)"hello, word!";
                 ^
error: warnings found and -Werror specified
1 error
1 warning
```

Usando Java

Observación

Para obtener información acerca de opciones adicionales y los argumentos usados con `-Xlint`, use las siguientes instrucciones, respectivamente:

```
$ javac --help -X
$ javac --help-lint
```

Clases, objetos y métodos

- En Java todos los métodos son declarados en una clase.
- Los **métodos estáticos (*static methods*)** no son invocados en objetos sino en clases.
- La ejecución del programa comienza en el método estático `main`.
- Los objetos `System.in`, `System.out` y `System.err` representan los *streams* para la entrada estándar, la salida estándar y el error estándar, respectivamente.
- Los **constructores (*constructors*)** son métodos empleados para inicializar los objetos cuando éstos son creados.

Clases, objetos y métodos

Ejemplo (procesando argumentos en la línea de comandos)

Mirar el archivo `oop/cli/java/CLI.java`.

```
$ cd oop/cli/java/
$ javac -Xlint CLI
$ java CLI a1 b1 c1
```

Clases, objetos y métodos

Ejemplo (números complejos)

Mirar los archivos
`oop/complex-numbers/java/Complex.java` y
`oop/complex-numbers/java/TestComplex.java`.

```
$ cd oop/complex-numbers/java/
$ javac -Xlint TestComplex.java
$ java TestComplex
```

Clases, objetos y métodos

Ejemplo (sistema de cuentas bancarias simplificado)

Una implementación en Java del sistema de cuentas bancarias simplificado se encuentra en el directorio `oop/sistema-cuentas-bancarias/java`.

Para ejecutar el programa use las siguientes instrucciones:

```
$ cd oop/toy-accounting-system/java
$ ant run
```

Interfaces e implementaciones

Descripción

*«Each class must have two parts: an interface and an implementation. . . The **interface** of a class is the one place where we assert all of the assumptions that a client may make about any instances of the class; the **implementation** encapsulates details about which no client may make assumptions.» (Booch, Maksimchuk, Engle, Young, Conallen y Houston 2007, pág. 51)*

Interfaces e implementaciones

Ejemplo (La interface *Collection*)

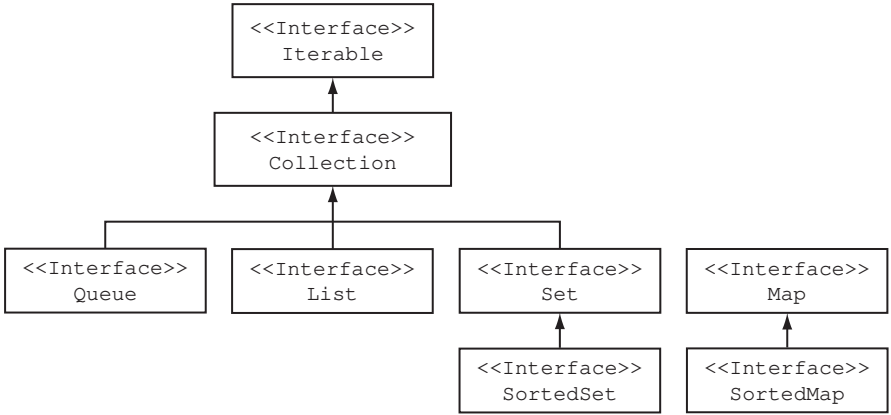


Figura 5.6

Interfaces e implementaciones

Descripción

Las interfaces de Java son un conjunto de declaraciones de métodos sin implementación.

Interfaces e implementaciones

Ejemplo (algunos métodos de la interfaz `Collection<T>`)

```
1  public Interface Collection<T> extends Iterable<T>{
2    boolean add(T e);
3    boolean addAll(Collection<T> c);
4    boolean contains(T e);
5    boolean containsAll(Collection<T> c);
6    boolean equals(Object o);
7    boolean isEmpty();
8    int size();
9    Iterator<T> iterator();
10   Stream<T> stream();
11 }
```

(continua en la próxima diapositiva)

Interfaces e implementaciones

Ejemplo (continuación)

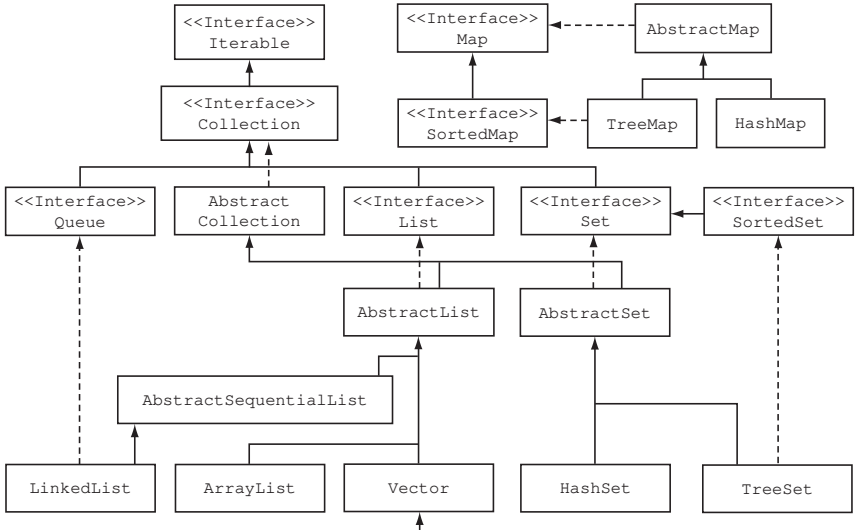
- La interfaz `Collection<T>` es definida usando polimorfismo paramétrico en donde el identificador `T` es un **parámetro de tipo (*type parameter*)**.
- En Java las variables cuyo tipo es un objeto usan una semántica por referencia y las clases son llamadas ***reference types***.
- Las colecciones pueden tener elementos de cualquier *reference types*:

```
List<String> listOfStrings;
Set<Integer> setOfInts;
```

- *«Java automatically wraps* `int` *values in* `Integer` *objects when they are added to a collection of integers, and automatically unwraps them when they are returned from a collection.»* (pág. 169)

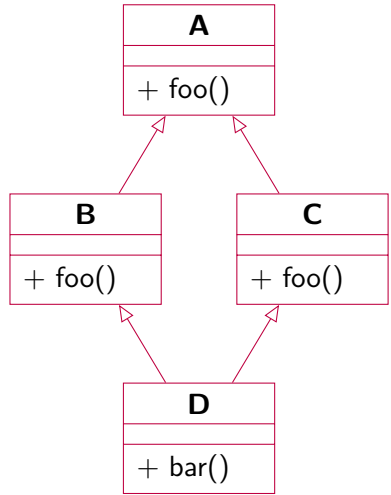
Interfaces e implementaciones

Ejemplo (Clases implementando algunas *collections*, Figura 5.7)



Herencia múltiple

*«The **diamond problem** is an ambiguity that arises when two classes `B` and `C` inherit from `A`, and class `D` inherits from both `B` and `C`. If there is a method in `A` that `B` and `C` have overridden, and `D` does not override it, then which version of the method does `D` inherit: that of `B`, or that of `C`?» (Wikipedia 2023-09-05)*



Interfaces e implementaciones

Ejemplo

Java no soporta herencia múltiple en las clases pero sí lo hace en las interfaces.
Mirar el directorio `oop/multiple-inheritance/java`.

```
$ cd oop/multiple-inheritance/java
$ make run
```

Interfaces e implementaciones

Lectura

Leer el blog [Kölling 2019].

Iteradores

Descripción

Cada colección Java define una forma de recorrer todos sus elementos en algún orden, es decir, cada colección define un **iterador**.

La superinterface `Iterable<T>` de la interface `Collection<T>` define un método `Iterator<T> iterator()`.

El tipo `Iterator<T>` es una interface con los siguientes métodos (pág. 173):

```
1 public Interface Iterator<T>{
2   public boolean hasNext(); // True if more elements, false otherwise
3   public T next();          // Returns the current element and advances
4   public void remove();     // Removes the current element from the store
5 }
```

Iteradores

Ejemplo (usando un iterador para procesar todos los elementos de una colección)

```
1 Collection<String> coll = ...;
2 Iterator<String> iter = coll.iterator();
3
4 while (iter.hasNext()) {
5   String element = iter.next();
6   Process element ...
7 }
```

Iteradores

Ejemplo (usando un *enhanced for loop* para procesar todos los elementos de una colección)

```
1 Collection<String> coll = ...;
2
3 for (String element : coll) {
4   Process element ...
5 }
```

Streams: map, filter, y reduce

Descripción

El tipo (clase) `Stream<T>` representa una secuencia de elementos de tipo `T`.

Streams: map, filter, y reduce

Descripción

El tipo (clase) `Stream<T>` representa una secuencia de elementos de tipo `T`.

Ejemplo

Mirar el archivo `oop/streams/StreamExamples.java`.

```
$ cd oop/streams
$ javac -Xlint StreamExamples.java
$ java StreamExamples
```

Referencias


Armstrong, Deborah J. (2006). The Quarks of Object-Oriented Development. Communications of the ACM 49.2, págs. 123-128. DOI: 10.1145/1113034.1113040 (vid. págs. 6, 7, 34-39).


Booch, Grady, Maksimchuk, Robert A., Engle, Michael W., Young, Bobbi J., Conallen, Jim y Houston, Kelli A. [1991] (2007). Object-Oriented Analysis and Design with Applications. 3.ª ed. Addison-Wesley (vid. págs. 34-39, 65).


Kölling, Michael (15 de jul. de 2019). The Evolving Nature of Java Interfaces. URL: https://blogs.oracle.com/javamagazine/post/the-evolving-nature-of-java-interfaces (vid. pág. 73).


Lee, Kent D. [2014] (2017). Foundations of Programming Languages. 2.ª ed. Undergraduate Topics in Computer Science. Springer (vid. pág. 53).


Louden, Kenneth C. y Lambert, Kenneth A. [1993] (2011). Programming Languages. Principles and Practice. 3.ª ed. Cengage Learning (vid. pág. 2).