# AMPLIACIÓN DE LA ONTOLOGÍA JLOO PARA MODELAR LA CLASE *MATH* DE JAVA®

GLORIA LUCÍA GIRALDO[1]
CARLOS OCAMPO[2]
IVÁN AMÓN[3]
CÉSAR LÓPEZ[4]
OMAR ESPINOSA[5]
CARLOS MARIO ZAPATA[6]

## Resumen

En este artículo, se extiende la ontología JLOO (Java® Learning Object Ontology) la cual sólo centra su dominio en los conceptos básicos del lenguaje Java®, incorporando otras librerías y clases. Como caso de estudio, se muestra la clase Math de la librería java.lang. El objetivo final, es ampliar la funcionalidad de JLOO, trasladando sus beneficios de aprendizaje hacia las librerías y clases.

---

[1] Ingeniera de Sistemas, Especialista en Ciencias Electrónicas e Informática, Magíster en *Thèorie et Ingénierie des Bases des Données* y Doctora en Informática, Profesora Asistente de la Escuela de Sistemas de la Universidad Nacional de Colombia, Sede Medellín.

[2] Ingeniero Electricista, Especialista en Telemática, Decano Programa Electromecánica y Electrónica INSTITUTO TECNOLÓGICO METROPOLITANO ITM, Medellín, Colombia.

[3] Ingeniero de Sistemas, Especialista en Técnicas Computarizadas de Producción, Docente Interno Universidad Pontificia Bolivariana UPB, Medellín, Colombia.

[4] Ingeniero de Sistemas, Especialista en Teleinformática, Docente Titular Universidad Pontificia Bolivariana UPB, Medellín, Colombia.

[5] Ingeniero Mecánico, EPM Telecomunicaciones, Medellín, Colombia.

[6] Ingeniero Civil, Especialista en Gerencia de Sistemas Informáticos, Magíster en Ingeniería de Sistemas y Doctor en Ingeniería, Profesor Asociado de la Escuela de Sistemas de la Universidad Nacional de Colombia, Sede Medellín.

Fecha de recepción: 22 de julio de 2008
Fecha de aceptación: 11 de agosto de 2008

**Palabras clave**

Ontología, JLOO, Java®, *Math, e-learning*

**Abstract**

The Java-Learning Object Ontology (JLOO) has, as a domain, the basic concepts of Java® language. In this paper, we extend JLOO to other libraries and classes. In order to explain the procedure, we use the *Math* class (belonging to the java.lang library) as a case study. As a result, we enhance the JLOO functionality with libraries and classes for teaching purposes.

**Key words**

Ontology, JLOO, Java®, *Math, e-learning*.

## Introducción

La enseñanza de los lenguajes de programación, es motivo de estudio por parte de varios investigadores, que abordan la temática desde diferentes puntos de vista. Algunos artículos, estudian los lenguajes de programación en general y desde su aspecto filosófico, mientras otros construyen ontologías alrededor de lenguajes de programación específicos. Para aportar al proceso de enseñanza-aprendizaje del lenguaje Java®, Ming-Che Lee *et al.,* propusieron la ontología que denominaron JLOO (*Java® Learning Oriented Ontology*) [1] partiendo de la clasificación hecha en CC2001 (Computing Curricula 2001 de la IEEE y la ACM). La ontología JLOO, sólo cubre aspectos atómicos básicos del lenguaje Java®, dejando por fuera las librerías frecuentemente utilizadas en tareas de programación, ya que la reutilización de los métodos incluidos en éstas, evita al programador tener que implementar múltiples funciones.

En este artículo, se extiende la ontología JLOO a nivel de clases, para que los docentes y estudiantes puedan transmitir y adquirir conocimiento adicional sobre los métodos que conforman las clases de las librerías, ampliando así el dominio de los objetos de aprendizaje. Se presenta el caso particular de la clase que contiene las funciones matemáticas de Java® (*Math*), pero el método utilizado se puede aplicar para extender la ontología a muchas otras librerías incorporadas o no en el lenguaje.

Un programador con conocimientos básicos podrá tomar la ontología y abordar los aspectos de conocimiento de la clase *Math* dentro de la jerarquía de subclases adicionada a JLOO.

La estructura de este artículo es la siguiente: en la segunda sección se presentan algunos trabajos realizados alrededor de la enseñanza de los lenguajes de programación y se detalla JLOO; la siguiente sección presenta conceptos sobre ontologías y sobre la clase *Math* de Java®; luego, se explica en detalle la metodología seguida para extender a JLOO con la clase *Math*; por último, se presentan las conclusiones y el posible trabajo futuro.

## Antecedentes

El estudio de la estructura de los lenguajes de programación, capturó la atención de varios investigadores en el mundo. Lando *et al.,* [2], jerarquizaron conceptos del dominio de los programas de computador. Turner y Eden, [3], profundizaron en este conocimiento desde la perspectiva de la filosofía de las ciencias de la computación y las ontologías en los lenguajes de programación, enfatizando la dimensión sintáctica de los programas. Estos mismos autores, [4], realizaron un análisis de los problemas en las ontologías de programas de computador. Turner, [5], documentó la influencia sobre la semántica de los lenguajes de programación del Platonismo/Formalismo.

Sosnovsky y Gavrilova en 2006, [6], presentaron un artículo más específico acerca del desarrollo de una ontología dirigida al tema de objetos de aprendizaje para el lenguaje de programación C, presentando un algoritmo para diseño de ontologías. La ontología construida recoge sus experiencias personales en la enseñanza del lenguaje C.

Ming-Che Lee *et al.,* [1], desarrollaron en 2005 la ontología "*Java® Learning Object Ontology,*" conocida como JLOO, partiendo de la clasificación hecha en CC2001 (*Computing Curricula* 2001 de la IEEE y la ACM) para el diseño curricular de los objetos de estudio de la ciencia de computación y, en especial, los conceptos sugeridos para los fundamentos de programación. Estos conceptos, son: modelos de datos, estructuras de control, orden de ejecución, encapsulamiento, relaciones entre componentes encapsulados y finalmente pruebas y depuración (véase apéndice 1). JLOO contribuye significativamente en la definición de unidades atómicas de conocimiento de los cursos introductorios de lenguaje Java®, en la construcción de unidades de conocimiento de JLOO compartibles y reutilizables, en la elaboración de diferentes estrategias de aprendizaje en un ambiente *E-learning* y en la definición de estrategias para el aprendizaje adaptativo.

Otros trabajos en el ámbito de los objetos de aprendizaje son el de Ruei-Yuan [7], el cual agrega elementos de lógica borrosa y técnicas de segmentación para descubrir la intención del usuario, y el de Neves y Adam [8], en el cual se construye una ontología llamada *"OntoRevPro"* para el aprendizaje colaborativo del lenguaje Java.

## JLOO

Por tratarse del trabajo que sirve de base a este artículo, a continuación se amplía el contenido de JLOO [1].

JLOO propone la organización de los objetos de aprendizaje Java® en un ambiente de cursos adaptativos. Un curso adaptativo, se puede definir como aquel que adapta, a las necesidades del aprendiz, la forma en que se transmite el objeto de aprendizaje. Los términos de JLOO, se basan en el *Computing Curricula* CC2001 de la ACM y la IEEE/CS. Estos términos se definen como unidades atómicas de conocimiento (que se pueden referir como objetos de aprendizaje) para el desarrollo de un curso introductorio de programación en Java®. JLOO contempla sólo las unidades de conocimiento básico sin incluir otras características más avanzadas de Java®, como el paquete *Swing*, la programación multihilos y la programación en red, entre otros.

La estructura de JLOO consiste en

**Definición de los objetos de aprendizaje:** se enmarca en la representación declarativa de conocimiento. Ésta incluye: hechos, conceptos, principios y modelos mentales.

**Las clases y la jerarquía de clases:** Los nodos hoja, en JLOO, son caracterizaciones de hechos. Se representan en plantillas que tienen definiciones de objetos de aprendizaje que conectan varios hechos. Los nodos que no son hojas representan principios, conceptos o modelos mentales. Las jerarquías que representa JLOO se basan en CC2001 y son las siguientes:

1. Modelos de datos: cubren los tipos básicos del lenguaje Java® y algunos conceptos simples del paradigma de la orientación a objetos. Incluyen, además, los conversores de tipo de datos, como elementos de la jerarquía que actúan sobre los datos. Se incluyen, también, los tipos de datos definidos por el usuario. La figura 1 presenta la jerarquía definida en JLOO para el modelo de datos.

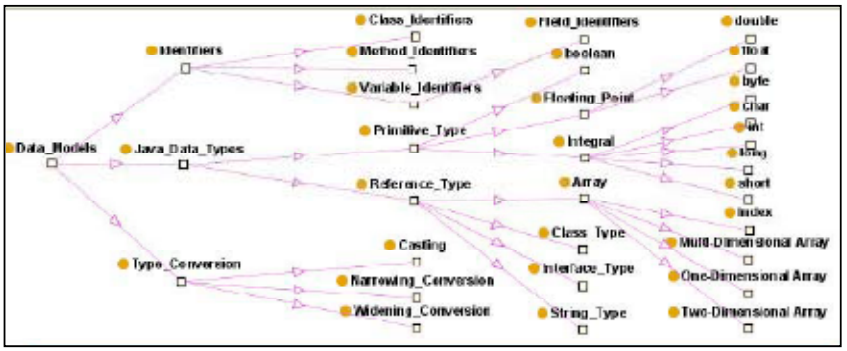


**Figura 1.** Jerarquía para el modelo de datos, tomada de [1]

2. Estructuras de control: Describen los efectos de aplicar las operaciones a los objetos del programa, esto es, qué hace una operación y cómo la hace. Se incluyen las instrucciones y expresiones en el lenguaje de programación. La figura 2 presenta la jerarquía definida en JLOO para las estructuras de control.

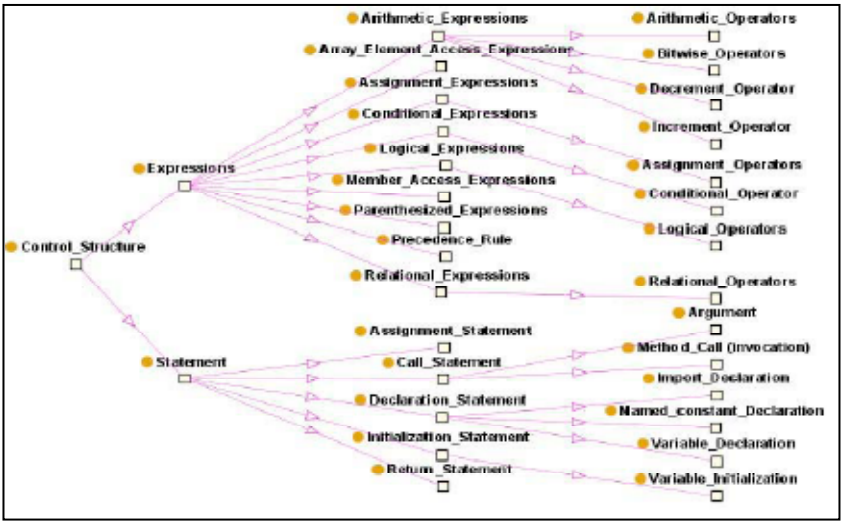


**Figura 2.** Jerarquía para las estructuras de control, tomada de [1]

3. Orden de ejecución: representa la manera en la cual se ejecutan las operaciones y puede ser diferente para tipos específicos de variables. La figura 3 presenta la jerarquía definida en JLOO para el orden de ejecución.

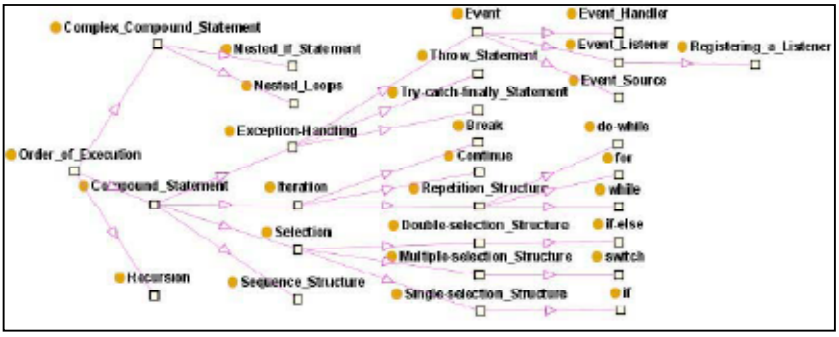


**Figura 3.** Jerarquía para el orden de ejecución, tomada de [1]

4. Encapsulamiento: en el paradigma de la programación orientada a objetos, el concepto de encapsulamiento se refiere a que el usuario de la clase puede obviar la implementación de los métodos y propiedades para concentrarse sólo en cómo usarlos. Por otro lado, se evita que el usuario pueda cambiar su estado de manera imprevista e incontrolada. La figura 4 presenta la jerarquía definida en JLOO para el encapsulamiento.

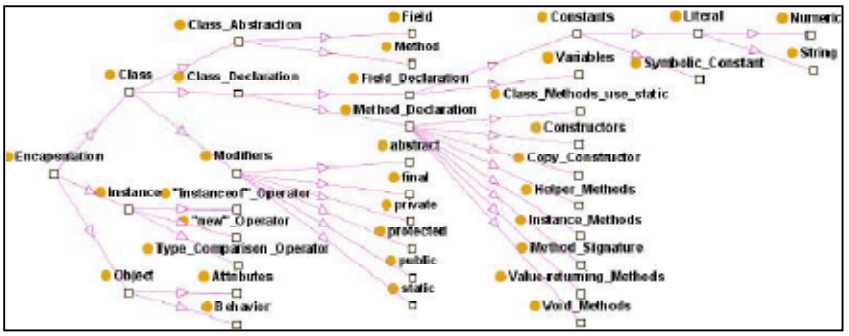


**Figura 4.** Jerarquía para encapsulamiento, tomada de [1]

5. Relaciones entre componentes encapsulados: son los conceptos de orientación a objetos que permiten intercambiar la información entre sus elementos, las responsabilidades de los componentes encapsulados con sus clientes y el mecanismo de herencia. La figura 5 presenta la jerarquía definida en JLOO para las relaciones entre los componentes encapsulados.

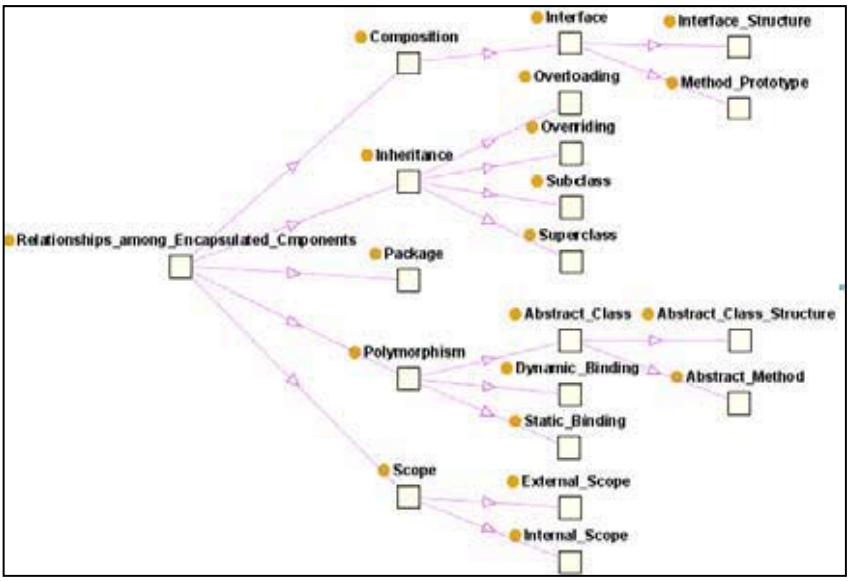


**Figura 5.** Jerarquía de relaciones entre componentes encapsulados, tomada de [1]

6. Pruebas y depuración: se extraen desde la Ingeniería de Software para construir este árbol. La figura 6 presenta la jerarquía definida en JLOO para pruebas y depuración.

## Marco teórico

### Construcción de ontologías

Una ontología busca reflejar la estructura de los conceptos de un dominio del mundo real. Se define como la representación explícita de un dominio o de un saber específico. Esta representación del dominio implica la identificación de conceptos, propiedades y atributos de los conceptos, restricciones sobre las propiedades y atributos y, aunque no siempre, los individuos. Un ejemplo de un dominio es el Lenguaje Java®. Actualmente, existen herramientas que permiten la construcción de ontologías, como Protégé 2000 y otras como OntoViz que permite su visualización.

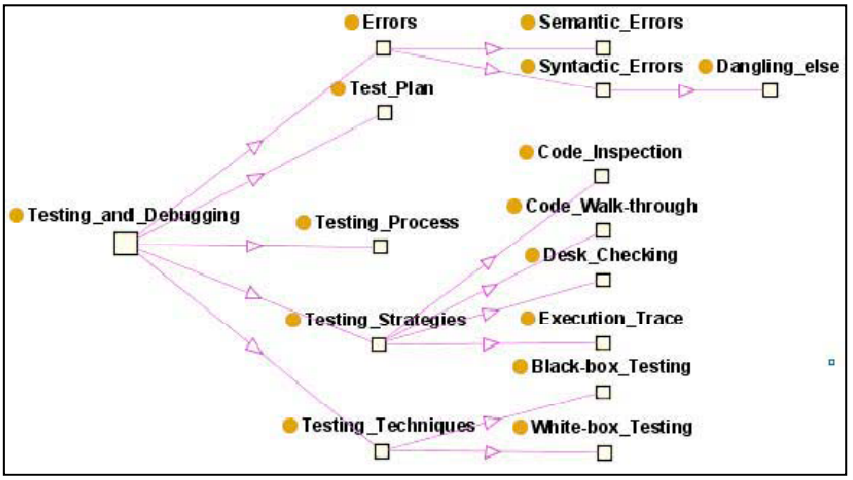


**Figura 6.** Jerarquía para pruebas y depuración, tomada de [1]

En [9], las autoras definen una metodología para la construcción de una ontología. Los pasos que ellas proponen son: 1. Definir el alcance de la ontología, 2. Considerar el reuso de ontologías existentes, 3. Enumerar los términos importantes, 4. Definir clases, 5. Definir propiedades (*slots*), 6. Definir características de las propiedades y 7. Crear instancias.

## La clase *Math* de Java®

El término *Math,* en el entorno de Java®, tiene dos significados: uno, la clase que se encuentra dentro de la librería Java.lang y otro, correspondiente a una librería propiamente definida. Para esta ontología, sólo se va a tratar la que se incluye en la librería Java.lang, que corresponde a las funciones matemáticas de mayor uso [10].

La clase *Math* contiene métodos para llevar a cabo operaciones matemáticas básicas como funciones exponenciales, logarítmicas y trigonométricas, entre otras. El apéndice 2 contiene todos los elementos de la clase *Math*.

### Ampliación de JLOO con la clase *MATH*

Para la construcción de la ontología, se siguen los pasos de la metodología propuesta en [9].

- **Paso 1:** Determinar el dominio y alcance de la ontología

  Para ello, se deben responder las preguntas:

  *¿Cuál es el dominio que cubrirá la ontología?*

  La ontología cubre el dominio de los objetos de aprendizaje para los fundamentos de programación en lenguaje Java®, incluyendo la clase *Math* de Sun® Microsystems.

  *¿Para qué se hace la ontología?*

  La ontología incorpora conceptos nuevos que no se incluyen en JLOO, permitiendo la adición de recursos para el aprendizaje de Java® y su aplicación en las matemáticas. Con esto, se extienden los beneficios de *e-Learning* para el aprendizaje de las funciones matemáticas incorporadas en el lenguaje Java®. Por otra parte, como valor añadido, la ontología permite establecer un método de reutilización de JLOO para integrar otras librerías, bien sea de las incorporadas en el *kit* de desarrollo de Java® (io, util y sql, entre otras), de otros proveedores o propias del programador.

  *¿Qué preguntas resuelve la ontología?*

  La ontología permite resolver las siguientes preguntas:

  - ¿Cuáles son las funciones que componen la clase *Math*?
  - ¿Cuáles son las funciones de la clase *Math* que retornan resultados de tipo Y = {*double, long, float, int*}?
  - ¿Cuáles son las funciones que componen una categoría o agrupación de funciones (trigonométricas, aritméticas, etc.)?

- ¿Cuáles son las funciones de una categoría o agrupación de funciones (trigonométricas, matemáticas, etc.), que retornan resultados de tipo Y = *{double, long, float, int}*?
- ¿De qué tipo son los posibles resultados retornados por una función determinada?
- ¿De qué tipo (*double, long, float, int*) es el resultado retornado por una función determinada?
- ¿Cuáles son las funciones que componen la clase *Math* con un número X = {0,1,2} de parámetros de entrada de un tipo Y = {*double, long, float, int*}?
- ¿Cuáles son las funciones de una categoría particular con un número X = {0,1,2} de parámetros de entrada de un tipo Y = {*double, long, float, int*}?

*¿Quién usará la ontología?*

La ontología se puede utilizar en los procesos de enseñanza y aprendizaje de fundamentos de programación en lenguaje Java® y herramientas informáticas para aplicaciones matemáticas. La pueden usar docentes que deseen presentar conceptos a sus estudiantes y estudiantes que deseen buscar respuesta a los interrogantes anteriores.

- **Paso 2:** Considerar la reutilización de ontologías existentes

El problema de la capacidad de uso y de reutilización aplicado al campo de las ontologías, establece que entre más capacidad de reutilización tenga una ontología menor capacidad de uso tiene, y viceversa [11]. Las ontologías de más alto nivel o *top-level*, las generales y las del dominio, capturan conocimiento de una manera independiente del problema que se desee resolver, mientras que las ontologías de métodos y las de tareas del dominio, se relacionan íntimamente con el conocimiento propio a la solución del problema. Todos estos tipos de ontologías, se pueden tomar de ontologías existentes en librerías como *DAML*

*Ontology Library, Ontolingua Ontology Library, Protégé Ontology Library, WebOnto Ontology Library, WebODE Ontology Library, SHOE Ontology Library*. Ellas también se pueden crear a partir de una combinación de ontologías de librerías diferentes. Sin embargo, Gómez-Pérez *et al.*, afirman que combinar ontologías no es una tarea fácil [12].

En este trabajo, se toma una ontología existente desde el año 2005 llamada *Java® Learning Object Ontology (*JLOO), la cual sólo se ocupa de los temas básicos de programación en Java y se construye una extensión de ella con el fin de permitir el manejo de las librerías y sus clases. La revisión bibliográfica no arrojó evidencia de otra ontología que extienda los conceptos del lenguaje Java® hacia las librerías.

- **Paso 3:** Enumerar términos importantes en la ontología

  JLOO, en su jerarquía "*Control Structure*", contiene el término *Control_Structure* del cual se deriva un brazo con los términos "*statement, call_statement, Method_call (invocation)*". Estos términos son importantes para orientar a quien desee agregar una nueva librería a JLOO y al profesor/alumno que estén participando del proceso de enseñanza-aprendizaje del lenguaje Java® y sus librerías. Los términos que se propone agregar a JLOO, y que se obtuvieron a partir del análisis de la estructura de la clase, son: *Language_Methods, Foreign_Methods, User_Methods, Java.lang, Math, Arithmetical_Functions, Trigonometrical_Functions, Fitting_Functions, Logarithmical_Functions, Random_Numbers_Functions, All(*)*[7], *Arc_Tangent, Sine, Cosine, Tangent, Hyperbolic_Cosine, Degrees_to_Radians, Radians_to_Degrees, Floor, Ceil, Qualified, Absolute_Value, Maximum, Minimun, Power_of, Square_Root, Cube_Root,* IEEE*_Remainder, Arc_Sine, Arc_Cosine, e_Rai-*

[7] Corresponde al asterisco (*)

*sed_to_the_power, Round, Natural_Logarithm, Base10_Logarithm, Random_Number, Data_Models, Java_Data_Types, Primitive_Type* e *Input_Parameter.*

- **Paso 4:** Definir las clases y la jerarquía de clases

  Se define una jerarquía para los términos identificados en el paso anterior, que se muestra en la figura 7.

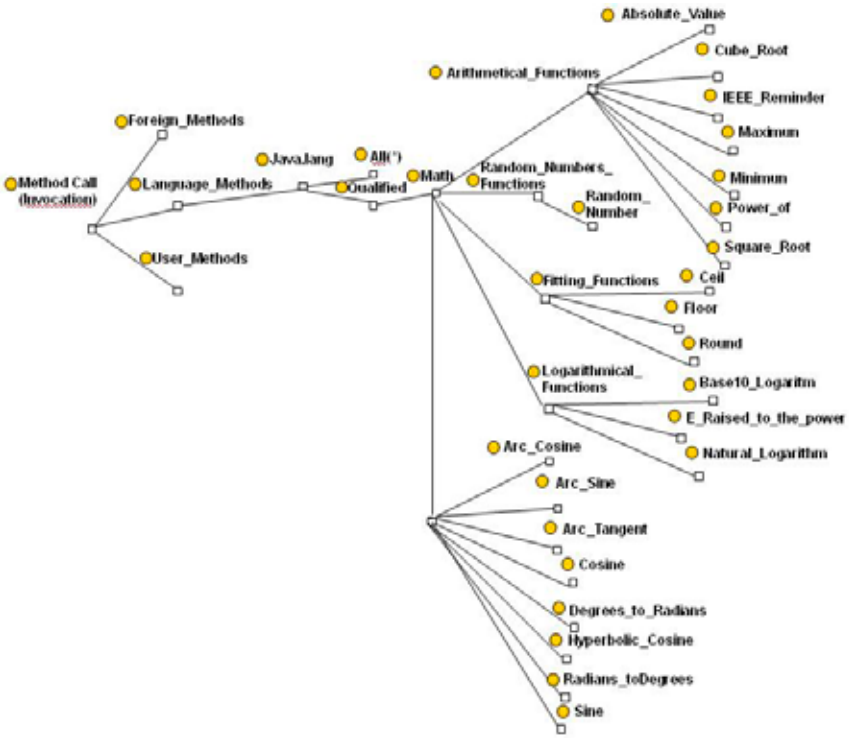


**Figura 7.** Jerarquía de clases adicionada a JLOO

La representación en Protégé, en su versión 3.1.1, se muestra en la figura 8. Por razones de simplicidad, se excluyen los conceptos de JLOO *Control_Structure, statement* y *call_statement.* A partir del concepto *Method_call (invocation),* se adicionan los

componentes que extienden la ontología. En primer término, se subdivide la invocación de métodos en tres ramas: métodos de otros fabricantes (*Foreign_Methods*), Métodos del lenguaje (*Language_Methods*) y métodos definidos por el usuario (*User_Methods*).


**Figura 8.** Representación en protégé de la clases adicionadas a JLOO

- Métodos de otros fabricantes: bajo la clase *Foreign_Methods,* se incorporaron librerías desarrolladas por otros fabricantes, diferentes a Sun® Microsystems, como por ejemplo JADE, JAI, entre otros.
- Métodos definidos por el usuario: bajo la clase *User_Methods*, se pueden incorporar librerías desarrolladas por el propio usuario.
- Métodos del lenguaje: bajo la clase *Language_Methods*, se incorporaron las diferentes librerías desarrolladas por Sun® Microsystems y que son parte del lenguaje Java®. Como ejemplo de la extensión de JLOO propuesta en este artículo, se presenta la jerarquía de la clase *Math* que pertenece a la librería java. lang. En forma similar, se podría extender la ontología a las otras librerías propias del lenguaje como java.io, java.util, java.swing, java.sql, entre otras. Se incluyó en la jerarquía el asterisco, como ALL(*), por tratarse de un concepto de interés en el aprendizaje del lenguaje, como es la inclusión de todas las clases de una librería.

  Para mejor entendimiento de la clase *Math*, se propone un agrupamiento de funciones, de acuerdo con el tipo de labor realizada. Por ejemplo, bajo el concepto de funciones de aproximación (*Fitting_Functions*), se reúnen diferentes funciones, que sirven para aproximar un valor a una determinada cantidad de decimales, esto es, la precisión de los valores numéricos (funciones piso, techo, redondeo).

  Los agrupamientos, corresponden a clases abstractas y los nodos hoja, correspondientes a los métodos finales por invocar, son clases concretas, ya que estos se instanciarán con las respectivas funciones del lenguaje (Ej. La clase abstracta "*Fitting_Functions*" se instancia tomando valores como "*Ceil*", "*Floor*" y "*Round*").

  La figura 9 presenta el esquema conceptual, con términos en español, utilizado en el diseño la ontología. Allí, se puede ver

cómo se relacionan los conceptos, sus atributos y las instancias, elementos que se deben definir en los pasos siguientes de la metodología. "Tipo Retornado", es una clase que representa un concepto importante dentro de la ontología, ya que en el lenguaje Java®, toda función o método tiene que retornar un resultado de un tipo de datos determinado. Aunque para el tipo *void* no se debe asociar explícitamente un valor, éste existe precisamente por la obligatoriedad que tiene la función o método de retornar algo de un tipo específico.

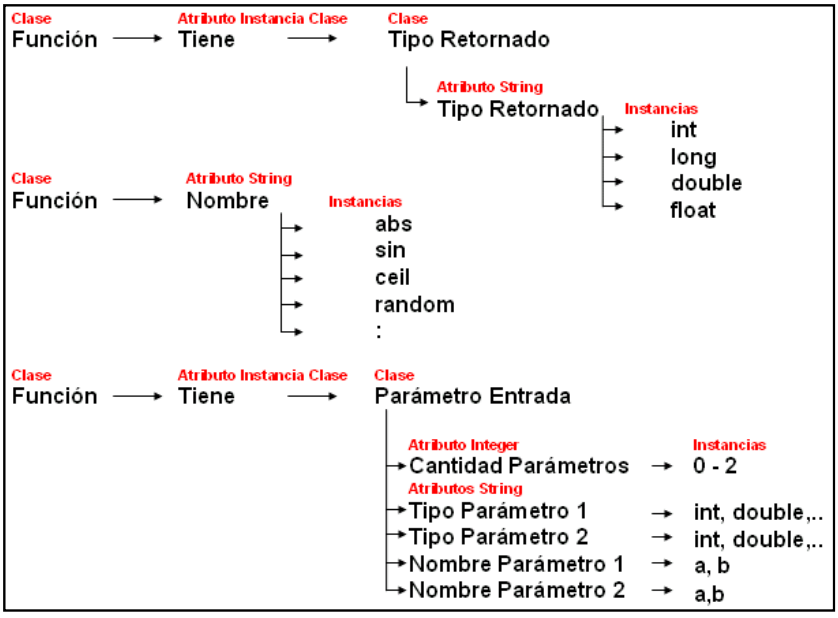


**FIGURA 9.** ESQUEMA CONCEPTUAL DE LA ONTOLOGÍA PROPUESTA

En la parte superior de la imagen, se muestra que una función tiene un "tipo retornado". Se define el atributo "tipo retornado" para asociar (instanciar) los posibles valores que puede tomar ese atributo. Para la clase *Math,* los tipos retornados permitidos son: *int, long, double y float*. Como "Función" y "Tipo Retornado" son clases, la manera de relacionarlas es través de un atributo tipo

"instancia de clase", llamado "Tiene". En la parte central de la imagen, se observa que una función tiene un atributo "nombre", y sus instancias corresponden a los nombres de las funciones o métodos de la clase *Math* (*abs, cos, sin, random,* …). La parte inferior de la imagen completa los elementos de las funciones, con la incorporación de los parámetros de entrada requeridos por éstas para su ejecución. Para la clase *Math* las funciones tienen 0, 1 ó 2 parámetros. "Parámetro Entrada" es una clase, porque también representa un concepto importante dentro del dominio de la ontología, ya que en el lenguaje Java®, toda función o método, puede o no tener parámetros de entrada. Se definen varios atributos, de acuerdo con la cantidad y con el tipo de parámetros que tienen las funciones o métodos de la clase. El atributo "Cantidad Parámetros" permite indicar (instanciar) cuántos parámetros de entrada requiere por cada función o método en particular. El atributo "Tipo Parámetro", indica (instancia) el tipo de cada parámetro. Como para la clase *Math* las funciones pueden tener hasta dos parámetros, entonces se enumeran con 1 y 2. De igual forma, el atributo "Nombre parámetro" indica el nombre local a la función de cada parámetro, y se enumeran con 1 y 2. La tabla 1 presenta ejemplos de las instancias requeridas para algunas funciones.

**Tabla 1.** Ejemplos de instancias de algunas funciones

| *Función* | *double random()* | *double cos (double a)* | *int max (int a, int b)* |
|---|---|---|---|
| Nombre | random | cos | Max |
| Tipo retornado | double | double | Int |
| Cantidad parámetros | 0 | 1 | 2 |
| Nombre parámetro 1 | | a | a |
| Nombre parámetro 2 | | | b |
| Tipo parámetro 1 | | Double | int |
| Tipo parámetro 2 | | | int |

- **Paso 5:** Definir las propiedades de las clases *(slots)*

  La tabla 2 explica las propiedades de las clases concretas de la ontología (funciones finales en la jerarquía).

**Tabla 2.** Propiedades de las clases concretas de la ontología

| *Nombre propiedad* | *Descripción* |
|---|---|
| *HasInputParameter* | Relaciona las clases función con la clase *Input_Parameter* mediante una instancia. |
| *HasReturnedType* | Relaciona las clases función con la clase *Primitive_Type*. |
| *Name* | Toda clase concreta debe tener un nombre a través del cual se invoca en el lenguaje. |
| *Number_of_Parameters* | Se refiere a la cantidad de parámetros de entrada que puede tener un método de la clase *Math*. |
| *Parameter1_Name* | Indica el nombre del primer parámetro. Depende del número de parámetros y es opcional. |
| *Parameter1_Type* | Indica el tipo de dato primitivo del primer parámetro. Depende del número de parámetros y es opcional. |
| *Parameter2_Name* | Indica el nombre del segundo parámetro. Depende del número de parámetros y es opcional. |
| *Parameter2_Type* | Indica el tipo de dato primitivo del segundo parámetro. Depende del número de parámetros y es opcional. |
| *Returned_Type* | Permite instanciar los tipos que puede devolver los métodos de la clase *Math*. |

- **Paso 6:** Definir las características *(facets)* de las propiedades *(slots)*

  Para ejemplificar, en la figura 10 se presentan las características definidas para una de las propiedades: "*HasInputParameter*", creadas mediante el editor de *slots* de Protégé. La tabla 3 resume las características de propiedades de las clases concretas de la ontología.

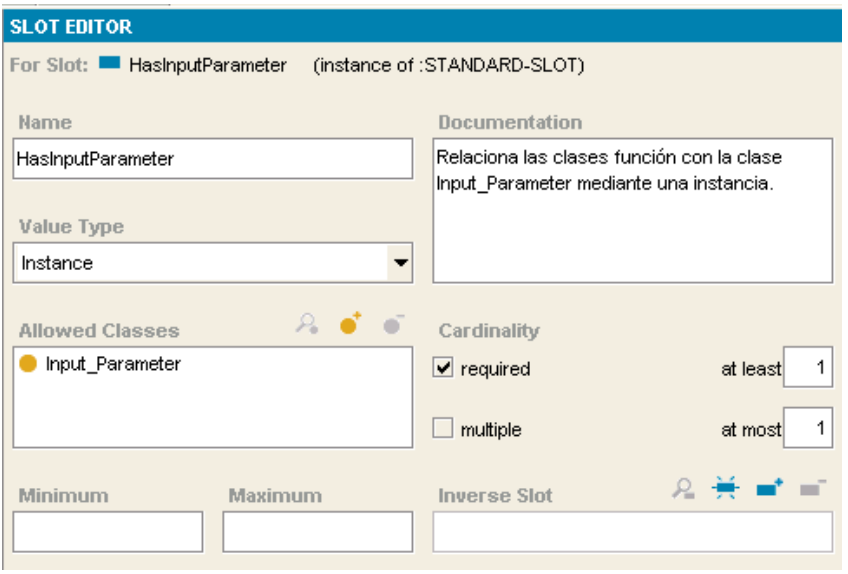


**Figura 10.** Características de la propiedad "*HasInputParameter*"

**Tabla 3.** Características de propiedades de las clases concretas de la ontología

| *Nombre propiedad* | *Tipo* | *Clases permitidas* | *Cardinalidad* |
|---|---|---|---|
| HasInputParameter | Instance | Input_Parameter | Required at least 1 |
| HasReturnedType | Instance | Primitive_Type | Required at least 1 |
| Name | String | | Required at least 1 |
| Number_of_Parameters | Integer | | Multiple at most 2 |
| Parameter1_Name | String | | Multiple at most 1 |
| Parameter1_Type | String | | Multiple at most 1 |
| Parameter2_Name | String | | Multiple at most 1 |
| Parameter2_Type | String | | Multiple at most 1 |
| Returned_Type | String | | Required at least 1 |

Se debe tener presente que el tipo *Instance* permite relacionar clases. Se debe aclarar que la característica cardinalidad de la propiedad *Number_of_Parameters* tiene como valor *Multiple*

*at most 2,* porque en la clase *Math* todas las funciones tienen máximo dos parámetros de entrada, lo cual puede no ser cierto para otro tipo de funciones.

- **Paso 7:** Crear instancias

La definición de las instancias se ilustrará con la clase concreta *Absolute_Value,* ya que es similar para todas las demás funciones. Se hizo de acuerdo con la definición del método en la clase *Math* de Java®. Este método, bajo el mismo nombre, puede retornar valores de cuatro tipos diferentes y recibir parámetros de entrada de cuatro tipos diferentes; en Java® se le denomina sobrecarga del método. Las figuras 11 a 13 ilustran la implementación de las instancias, con el mismo nombre, pero distintos parámetros de entrada, y distintos tipos de valor retornado.

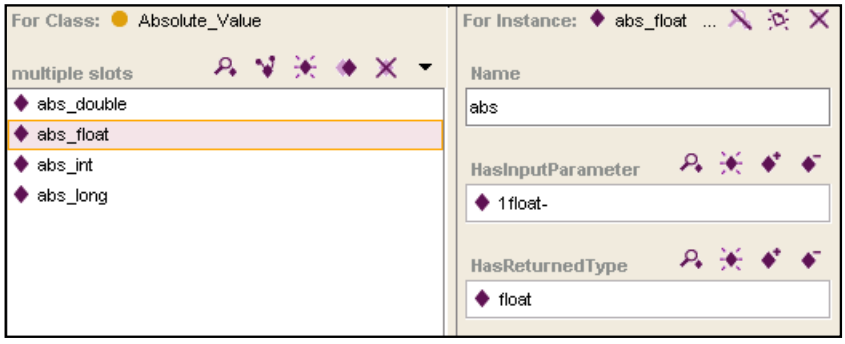


**Figura 11.** Instancias para la clase *Absolute_Value* de tipo *float*

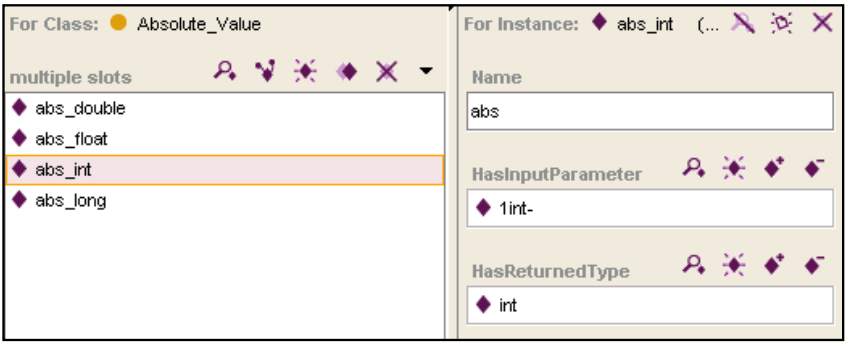


**Figura 12.** Instancias para la clase *Absolute_Value* de tipo *int*

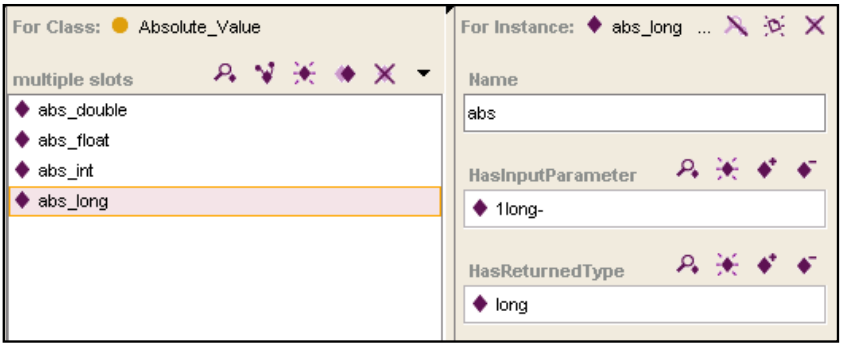


**Figura 13.** Instancias para la clase *Absolute_Value* de tipo *long*

## Conclusiones

En este artículo se propone una extensión a la ontología JLOO, incorporándole la clase *Math* de la librería java.lang, con el fin de ampliar el dominio de los objetos de aprendizaje. La reutilización de ontologías existentes tiene ventajas, pero también desventajas, ya que el dominio puede no adecuarse exactamente al requerido y presentar términos o conceptos sobrantes o faltantes.

Para realizar un trabajo similar a éste, ampliando la ontología a otras librerías, se requiere total dominio de las características de las funciones por incluir y, por tanto, no se debe emprender esta

labor si no se cumple este requisito. Asimismo, es deseable contar con experiencia docente en la enseñanza de los temas que se están plasmando en la ontología.

Se recomienda a los novatos en construcción de ontologías, seguir una metodología propuesta para este fin, como la que se propone en [9]. Aun así, durante el proceso, se debe tener cuidado de hacer coincidir las preguntas inicialmente concebidas, con las que la ontología implementada logra responder finalmente.

Las ontologías desempeñan un papel muy importante en el proceso de enseñanza-aprendizaje, especialmente en cuanto a procesos de *e-learning*.

### Trabajo futuro

El método utilizado en este artículo se puede aplicar para incluir las clases de otras librerías propias del lenguaje (java.io, java.util, java.sql, entre otras), clases de librerías de proveedores diferentes a Sun® Microsystems (JADE, JAI, entre otras) y/o clases de librerías propias del usuario, dotando a docentes y estudiantes de una completa herramienta para la enseñanza y el aprendizaje del lenguaje Java®.

Quedaría por demostrar, de manera experimental, cuáles son los beneficios de usar la ontología JLOO, con las ampliaciones que se proponen en este artículo, en los procesos de enseñanza-aprendizaje del lenguaje Java®.

Además del uso de la ontología JLOO, complementada con otras clases, el trabajo futuro se puede encaminar a la construcción de las ontologías de otros lenguajes orientados a objetos como C++ y C#. También, se podrían determinar las similitudes entre estos lenguajes, con el fin de proponer una ontología general para los lenguajes orientados a objetos, como una especie de metaontología.

## Referencias bibliográficas

[1] Ming-Che Lee; Ding Yen Ye; Tzone I Wang. (2005). "Java learning object ontology". ICALT 2005. Fifth IEEE International Conference on Advanced Learning Technologies, pp. 538-542.

[2] Lando Pascal, Lapujade Anne, Kassel Gilles, Furst Frédéric (2008).“Towards a general ontology of computer programs”. Seleccionado para ICSOFF 2008. 3rd International Conference on Software and Data technologies.

[3] Raymond Turner, Amnon H. Eden. (2007).“Towards a Programming Language Ontology.” Ch. In: Gordana Dodig-Crnkovic, Susan Stuart (ed.), *Computing, Philosophy, and Cognitive Science*, Cambridge, UK: Cambridge Scholars Press.

[4] Amnon H Eden, Raymond Turner. (2007) “Problems in the Ontology of Computer Programs.” Applied Ontology, Vol. 2, No. 1, pp. 13-36. Amsterdam: IOS Press. Amsterdam, The Netherlands: IOS Press.

**[5]** Raymond Turner. (2007). “Understanding Programming Languages” Minds and Machines, Vol. 17, Issue 2, pp. 203-216. Kluwer Academic Publishers Hingham, MA, USA.

[6] Sosnovsky Sergey, Gavrilova Tatiana. (2006). “Development of educational ontology for c-Programming”. International Journal Information Theories & Applications, Vol. 13, Number 4, pp 303-308.

[7] Ruei-Yuan Guo. (2005) “Mining User Intension with Fuzzy Theory and Clustering Technique for Learning Object Content Recommendation of e-Learning Systems “. Tesis (Maestría en Computer Science and Engineering). University Sun Yat-sen. Disponible en: http://etd.lib.nsysu.edu.tw/ETD-db/ETD-search/view_etd?URN=etd-0908106-111204.

[8] Neves Maria de Fátima, Adan C. Juan Manuel. (2006). *“OntoRevPro: Uma ontología sobre Revisão de Programas para o Aprendizado Colaborativo de Programação em Java”.* SBIE 2006. xvii Simpósio Brasileiro de Informática na Educação.

[9] Noy, Natalya, McGuinness Deborah. “Ontology Development 101: A Guide to Creating Your First Ontology”. Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, March 2001.

[10] *Java™ 2 Platform Std. Ed. v1.4.2 - Class Math.* [En línea]. s.p.i. http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Math.html [Consulta: 24 Feb. 2008].

[11] Klinker G, Bhola C., Dallemagne G., Marquez D. McDermott J. (1991) Usable and reusable programming constructs. Knowledge Acquisition 3:117-136

[12] Asunción Gómez Pérez, Mariano Fernández López, Oscar Corcho (2004). Ontological Engineering: with examples from the areas of knowledge, e-commerce and Semantic Web, Springer-Verlag, London.

**Apéndice 1.** Conceptos que debe cubrir un currículo introductorio según el CC2001

## Programming fundamentals

| *Concept* | *Description* | *Associated activities* |
|---|---|---|
| Data models | Standard structures for representing data; abstract (described by an implementation) descriptions. | Read and explain values of program objects; create, implement, use, and modify programs that manipulate standard data structures. |
| Control structures | Effects of applying operations to program objects; what an operation does (described by a model); how an operation does it (described by an implementation). | Read and explain the effects of operations; implement and describe operations; construct programs to implement a range of standard algorithms. |
| Order of execution | Standard control structures: sequence; selection, iteration; functions calls and parameter passing. | Make appropriate use of control structures in the design of algorithms and then implement those structures in executable programs. |
| Encapsulation | Indivisible bundling of related entities; client view based on abstraction and information-hiding; implementer view based on internal detail. | Use existing encapsulated components in programs; design, implement, and document encapsulated components. |
| Relationships among encapsulated components | The role of interfaces in mediating information exchange; responsibilities of encapsulated components to their clients; the value of inheritance. | Explain and make use of inheritance and interface relationships; incorporate inheritance and interfaces into the design and implementation of programs. |
| Testing and debugging | The importance of testing; debugging strategies. | Design effective tests; identify and correct coding and logic errors. |

## Apéndice 2. Clase *Math*

### Atributos:

| | |
|---|---|
| static double | **E** The double value that is closer than any other to *e*, the base of the natural logarithms. |
| static double | **PI** The double value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter. |

Operaciones (métodos):

| | |
|---|---|
| static double | **abs**(double a) Returns the absolute value of a double value. |
| static float | **abs**(float a) Returns the absolute value of a float value. |
| Static int | **abs**(int a) Returns the absolute value of an int value. |
| static long | **abs**(long a) Returns the absolute value of a long value. |
| static double | **acos**(double a) Returns the arc cosine of a value; the returned angle is in the range 0.0 through *pi*. |
| static double | **asin**(double a) Returns the arc sine of a value; the returned angle is in the range -*pi*/2 through *pi*/2. |
| static double | **atan**(double a) Returns the arc tangent of a value; the returned angle is in the range -*pi*/2 through *pi*/2. |
| static double | **atan2**(double y, double x) Returns the angle *theta* from the conversion of rectangular coordinates (x, y) to polar coordinates (r, *theta*). |
| static double | **cbrt**(double a) Returns the cube root of a double value. |
| static double | **ceil**(double a) Returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer. |
| static double | **copySign**(double magnitude, double sign) Returns the first floating-point argument with the sign of the second floating-point argument. |
| static float | **copySign**(float magnitude, float sign) Returns the first floating-point argument with the sign of the second floating-point argument. |
| static double | **cos**(double a) Returns the trigonometric cosine of an angle. |
| static double | **cosh**(double x) Returns the hyperbolic cosine of a double value. |
| static double | **exp** (double a) Returns Euler's number *e* raised to the power of a double value. |
| static double | **expm1**(double x) Returns $e^x$ -1. |

→

| | |
|---|---|
| static double | **floor**(double a) Returns the largest (closest to positive infinity) double value that is less than or equal to the argument and is equal to a mathematical integer. |
| Static int | **getExponent**(double d) Returns the unbiased exponent used in the representation of a double. |
| Static int | **getExponent**(float f) Returns the unbiased exponent used in the representation of a float. |
| static double | **hypot**(double x, double y) Returns sqrt($x^2$ +$y^2$) without intermediate overflow or underflow. |
| static double | **IEEEremainder**(double f1, double f2) Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard. |
| static double | **log**(double a) Returns the natural logarithm of a double value. |
| static double | **log10**(double a) Returns the base 10 logarithm of a double value. |
| static double | **log1p**(double x) Returns the natural logarithm of the sum of the argument and 1. |
| static double | **max**(double a, double b) Returns the greater of two double values. |
| static float | **max**(float a, float b) Returns the greater of two float values. |
| static int | **max**(int a, int b) Returns the greater of two int values. |
| static long | **max**(long a, long b) Returns the greater of two long values. |
| static double | **min**(double a, double b) Returns the smaller of two double values. |
| static float | **min**(float a, float b) Returns the smaller of two float values. |
| static int | **min**(int a, int b) Returns the smaller of two int values. |
| static long | **min**(long a, long b) Returns the smaller of two long values. |
| static double | **nextAfter**(double start, double direction) Returns the floating-point number adjacent to the first argument in the direction of the second argument. |
| static float | **nextAfter**(float start, double direction) Returns the floating-point number adjacent to the first argument in the direction of the second argument. |
| static double | **nextUp**(double d) Returns the floating-point value adjacent to d in the direction of positive infinity. |
| static float | **nextUp**(float f) Returns the floating-point value adjacent to f in the direction of positive infinity. |
| static double | **pow**(double a, double b) Returns the value of the first argument raised to the power of the second argument. |

→

| | |
|---|---|
| static double | **random**() Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0. |
| static double | **rint**(double a) Returns the double value that is closest in value to the argument and is equal to a mathematical integer. |
| static long | **round**(double a) Returns the closest long to the argument. |
| static int | **round**(float a) Returns the closest int to the argument. |
| static double | **scalb**(double d, int scaleFactor) Return $d \times 2^{scaleFactor}$ rounded as if performed by a single correctly rounded floating-point multiply to a member of the double value set. |
| static float | **scalb**(float f, int scaleFactor) Return $f \times 2^{scaleFactor}$ rounded as if performed by a single correctly rounded floating-point multiply to a member of the float value set. |
| static double | **signum**(double d) Returns the signum function of the argument; zero if the argument is zero, 1.0 if the argument is greater than zero, -1.0 if the argument is less than zero. |
| static float | **signum**(float f) Returns the signum function of the argument; zero if the argument is zero, 1.0f if the argument is greater than zero, -1.0f if the argument is less than zero. |
| static double | **sin**(double a) Returns the trigonometric sine of an angle. |
| static double | **sinh**(double x) Returns the hyperbolic sine of a double value. |
| static double | **sqrt**(double a) Returns the correctly rounded positive square root of a double value. |
| static double | **tan**(double a) Returns the trigonometric tangent of an angle. |
| static double | **tanh**(double x) Returns the hyperbolic tangent of a double value. |
| static double | **toDegrees**(double angrad) Converts an angle measured in radians to an approximately equivalent angle measured in degrees. |
| static double | **toRadians**(double angdeg) Converts an angle measured in degrees to an approximately equivalent angle measured in radians. |
| static double | **ulp**(double d) Returns the size of an ulp of the argument. |
| static float | **ulp**(float f) Returns the size of an ulp of the argument. |