Inheritance

Prof. Michele Loreti

Programmazione Avanzata
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

Class Employee (1/2)

```java
public class Employee {

  private final String name;
  private double salary;
  private static int lastId = 0;
  private int id;

  public Employee( String name , double salary ) {
    Employee.lastId++;
    this.id = lastId;
    this.name = name;
    this.salary = salary;
  }

  public void raiseSalary( double byPercent ) {
    double raise = salary*byPercent/100;
    salary += raise;
  }
```

Class Employee (2/2)

```java
  public void setSalary( double salary ) {
    this.salary = salary;
  }

  public double getSalary( ) {
    return this.salary;
  }

  public String getName() {
    return this.name;
  }

  public int getId() {
    return id;
  }
}
```

Extending a class

Let us define a new class Manager retaining some functionalities of the Employee but specifying how managers are different.

Extending a class

UNICAM Università di Camerino 1336

Let us define a new class Manager retaining some functionalities of the Employee but specifying how managers are different.

```
public class Manager extends Employee {
   ...//Added fields
   ...//added or overriding methods
}
```

Extending a class

Let us define a new class Manager retaining some functionalities of the Employee but specifying how managers are different.

```
public class Manager extends Employee {
  ...//Added fields
  ...//added or overriding methods
}
```

The keywork extends is used to indicate that a new class is defined that derives from an existing class.

Extending a class

Let us define a new class Manager retaining some functionalities of the Employee but specifying how managers are different.

```
public class Manager extends Employee {
  ...//Added fields
  ...//added or overriding methods
}
```

The keyword extends is used to indicate that a new class is defined that derives from an existing class.

The existing class is called superclass while the new one is called subclass.

Extending a class

Let us define a new class Manager retaining some functionalities of the Employee but specifying how managers are different.

```
public class Manager extends Employee {
  ...//Added fields
  ...//added or overriding methods
}
```

The keyword extends is used to indicate that a new class is defined that derives from an existing class.

The existing class is called superclass while the new one is called subclass.

A subclass have more functionalities than their superclasses!

Defining and Inheriting Subclass Methods

Our Manager class have a new instance variable to store the bonus and a new method to set it:

Defining and Inheriting Subclass Methods

Our Manager class have a new instance variable to store the bonus and a new method to set it:

```
public class Manager extends Employee {
  private double bonus;
  ...
  public void setBonus(double bonus) {
    this.bonus = bonus;
  }
}
```

Defining and Inheriting Subclass Methods

Our Manager class have a new instance variable to store the bonus and a new method to set it:

```
public class Manager extends Employee {
  private double bonus;
  ...
  public void setBonus(double bonus) {
    this.bonus = bonus;
  }
}
```

On a Manager object we can invoke the setBonus but also all the methods defined in Employee:

Defining and Inheriting Subclass Methods

Our Manager class have a new instance variable to store the bonus and a new method to set it:

```
public class Manager extends Employee {
  private double bonus;
  ...
  public void setBonus(double bonus) {
    this.bonus = bonus;
  }
}
```

On a Manager object we can invoke the setBonus but also all the methods defined in Employee:

```
Manager boss = new Manager( ... );
boss.setBonus(10000);
boss.raiseSalary(5);
```

Method Overriding

Sometimes, a subclass modifies methods defined in the superclass.

Method Overriding

Sometimes, a subclass modifies methods defined in the superclass.

Example: the salary of a Manager is computed by adding the bonus to the salary of an Employee

Method Overriding

Sometimes, a subclass modifies methods defined in the superclass.

Example: the salary of a Manager is computed by adding the bonus to the salary of an Employee

```
public class Manager extends Employee {
  ...
  public double getSalary( ) {
    return super.getSalary()+this.bonus;
  }
  ...
}
```

Method Overriding

```
public class Employee {

  public boolean worksFor( Employee supervisor ) {
    ...
  }

}

public class Manager extends Employee {

  public boolean worksFor( Manager supervisor ) {
    ...
  }

}
```

Method Overriding

```
public class Employee {

  public boolean worksFor( Employee supervisor ) {
    ...
  }

}

public class Manager extends Employee {

  public boolean worksFor( Manager supervisor ) {
    ...
  }

}
```

This is not overriding! A new method is defined!

Method Overriding

```
public class Employee {

  public Employee getSupervisor( ) {
    ...
  }

}

public class Manager extends Employee {

  public Manager getSupervisor( ) {
    ...
  }

}
```

Method Overriding

```
public class Employee {

  public Employee getSupervisor( ) {
    ...
  }

}

public class Manager extends Employee {

  public Manager getSupervisor( ) {
    ...
  }

}
```

This is overriding!

Method Overriding

UNICAM Università di Camerino 1336

```
public class Employee {

  public Employee getSupervisor( ) {
    ...
  }

}

public class Manager extends Employee {

  public Manager getSupervisor( ) {
    ...
  }

}
```

This is overriding!

The use of @Override is strongly recommended!

Subclass Construction

A subclass must invoke the appropriate constructor of its superclass to fill the `private` fields:

Subclass Construction

A subclass must invoke the appropriate constructor of its superclass to fill the private fields:

```
public Manager( String name, double salary) {
  super(name,salary);
  this.bonus = 0;
}
```

Subclass Construction

A subclass must invoke the appropriate constructor of its superclass to fill the private fields:

```
public Manager( String name, double salary) {
  super(name,salary);
  this.bonus = 0;
}
```

If no super constructor is invoked, the superclass must have a default constructor that is called implicitly.

Superclass Assignment

It is legal to assign an object from a subclass to a variable whose type is a superclass:

Superclass Assignment

It is legal to assign an object from a subclass to a variable whose type is a superclass:

```
Manager boss = new Manager(...);
Employee empl = boss;
```

Superclass Assignment

UNICAM
Università di Camerino
1336

It is legal to assign an object from a subclass to a variable whose type is a superclass:

```
Manager boss = new Manager(...);
Employee empl = boss;
```

Question: what happens when the following code is executed?

Superclass Assignment

It is legal to assign an object from a subclass to a variable whose type is a superclass:

```
Manager boss = new Manager(...);
Employee empl = boss;
```

Question: what happens when the following code is executed?

```
empl.getSalary();
```

Superclass Assignment

It is legal to assign an object from a subclass to a variable whose type is a superclass:

```
Manager boss = new Manager(...);
Employee empl = boss;
```

Question: what happens when the following code is executed?

```
empl.getSalary();
```

The dynamic type of a receiver is used to select the method to invoke (dynamic method lookup)!

Example...

```
Employee[] staff = new Employee[...];
staff[0] = new Employee(...);
staff[1] = new Manager(...);
staff[2] = new Employee(...);
...
double sum = 0;
for (Employee e: staff) {
  sum += e.getSalary();
}
```

Example...

```
Employee[] staff = new Employee[...];
staff[0] = new Employee(...);
staff[1] = new Manager(...);
staff[2] = new Employee(...);
...
double sum = 0;
for (Employee e: staff) {
  sum += e.getSalary();
}
```

Thanks to dynamic method lookup, the right version of getSalary() is selected!

Cast

Let us consider the following code:

Cast

Let us consider the following code:

```
Employee empl = new Manager(...);
empl.setBonus(10000);
```

Cast

Let us consider the following code:

```
Employee empl = new Manager(...);
empl.setBonus(10000);
```

If really needed, we can use instanceof and explicit cast to access to methods of a subclass:

Cast

Let us consider the following code:

```
Employee empl = new Manager(...);
empl.setBonus(10000);
```

If really needed, we can use instanceof and explicit cast to access to methods of a subclass:

```
if (empl instanceof Manager) {
  Manager mgr = (Manager) empl;
  mgr.setBonus(10000);
}
```

Final methods

When a method is declared `final`, no subclass can override it:

Final methods

When a method is declared final, no subclass can override it:

```
public class Employee {
  ...
  public final String getName() {
    return this.name;
  }
}
```

Final methods

When a method is declared final, no subclass can override it:

```
public class Employee {
  ...
  public final String getName() {
    return this.name;
  }
}
```

An example of final method is the getClass method of Object.

Final methods

When a method is declared `final`, no subclass can override it:

```
public class Employee {
  ...
  public final String getName() {
    return this.name;
  }
}
```

An example of `final` method is the `getClass` method of `Object`.

Modifier `final` can be applied also to classes to prevent others from subclassing:

Final methods

When a method is declared final, no subclass can override it:

```
public class Employee {
  ...
  public final String getName() {
    return this.name;
  }
}
```

An example of final method is the getClass method of Object.

Modifier final can be applied also to classes to prevent others from subclassing:

```
public final class Executive extends Manager {
  ...
}
```

Abstract Methods and Classes

A class can define a method without an implementation, forcing subclasses to implement it.

Abstract Methods and Classes

A class can define a method without an implementation, forcing subclasses to implement it.

This method, and the class that contains it, are called abstract.

Abstract Methods and Classes

A class can define a method without an implementation, forcing subclasses to implement it.

This method, and the class that contains it, are called abstract.

```
public abstract class Person {
  private final String name;

  public Person( String name ) {
    this.name = name;
  }

  public final String getName() {
    return name;
  }

  public abstract int getId();
}
```

Abstract Methods and Classes

```
public class Student extends Person {
  private final int id;

  public Student( int id, String name ) {
    super( name );
    this.id = id;
  }

  public final int getId() {
    return id;
  }


}
```

Protected Access

```
public class Employee {
  protected double salary;
  ...
}


public class Manager {
  ...

  public double getSalary() {
    return this.salary+this.bonus;
  }
  ...
}
```

Protected Access

```
public class Employee {
  protected double salary;
  ...
}


public class Manager {
  ...

  public double getSalary() {
    return this.salary+this.bonus;
  }
  ...
}
```

Protected fields must be used with caution! Protected methods and constructor are more standard!

Inheritance and Default Methods

```
public interface Named {
  public String getName() { return ""; }
}

public class Person implements Named {
  ...
  public String getName() { return this.name; }
  ...
}

public class Student extends Person implements Named {
  ...
}
```

Inheritance and Default Methods

```
public interface Named {
  public String getName() { return ""; }
}

public class Person implements Named {
  ...
  public String getName() { return this.name; }
  ...
}

public class Student extends Person implements Named {
  ...
}
```

In this case we have not a conflict! The class-win approach is used (to guarantee backward compatibility)!

Method Expressions

Method Expressions

```java
public class Worker {
  public void work() {
    for( int i=0 ; i<100; i++ ) {
      System.out.println("Working...");
    }
  }
}
```

Method Expressions

```java
public class Worker {
  public void work() {
    for( int i=0 ; i<100; i++ ) {
      System.out.println("Working...");
    }
  }
}
```

```java
public class ConcurrentWorker {
  public void work() {
    Thread t = new Thread(super::work);
    t.start();
  }
}
```

Object: The Cosmic Superclass!

Every Java class directly or indirectly extends the class Object:

```
public class Employee { ... }
```

is equivalent to

```
public class Employee extends Object { ... }
```

Object: The Cosmic Superclass!

Every Java class directly or indirectly extends the class Object:

```
public class Employee { ... }
```

is equivalent to

```
public class Employee extends Object { ... }
```

Class Object contains the following methods:

Object: The Cosmic Superclass!

Every Java class directly or indirectly extends the class Object:

```
public class Employee { ... }
```

is equivalent to

```
public class Employee extends Object { ... }
```

Class Object contains the following methods:

- String toString()

Object: The Cosmic Superclass!

Every Java class directly or indirectly extends the class Object:

```
public class Employee { ... }
```

is equivalent to

```
public class Employee extends Object { ... }
```

Class Object contains the following methods:

- String toString()
- boolean equals(Object other)

Object: The Cosmic Superclass!

Every Java class directly or indirectly extends the class Object:

```
public class Employee { ... }
```

is equivalent to

```
public class Employee extends Object { ... }
```

Class Object contains the following methods:

- String toString()
- boolean equals(Object other)
- int hashCode()

Object: The Cosmic Superclass!

Every `Java` class directly or indirectly extends the class Object:

```
public class Employee { ... }
```

is equivalent to

```
public class Employee extends Object { ... }
```

Class Object contains the following methods:

- String toString()
- boolean equals(Object other)
- int hashCode()
- Class<?> getClass()

Object: The Cosmic Superclass!

Every Java class directly or indirectly extends the class Object:

```
public class Employee { ... }
```

is equivalent to

```
public class Employee extends Object { ... }
```

Class Object contains the following methods:

- `String toString()`
- `boolean equals(Object other)`
- `int hashCode()`
- `Class<?> getClass()`
- `protected Object clone()`

Object: The Cosmic Superclass!

Every Java class directly or indirectly extends the class Object:

```
public class Employee { ... }
```

is equivalent to

```
public class Employee extends Object { ... }
```

Class Object contains the following methods:

- String toString()
- boolean equals(Object other)
- int hashCode()
- Class<?> getClass()
- protected Object clone()
- protected void finalize ()

Object: The Cosmic Superclass!

Every Java class directly or indirectly extends the class Object:

```
public class Employee { ... }
```

is equivalent to

```
public class Employee extends Object { ... }
```

Class Object contains the following methods:

- `String toString()`
- `boolean equals(Object other)`
- `int hashCode()`
- `Class<?> getClass()`
- `protected Object clone()`
- `protected void finalize ()`
- `wait, notify, notifyAll`

Method toString

Method `toString` is used to obtain a string representation of an object:

Method toString

Method toString is used to obtain a string representation of an object:

```
public class Employee {
  ...
  public String toString() {
    return getClass().getName()+"[name="+this.name+
      ",salary="+this.salary+"]";
  }
  ...
}
```

Method toString

Method toString is used to obtain a string representation of an object:

```
public class Employee {
  ...
  public String toString() {
    return getClass().getName()+"[name="+this.name+
      ",salary="+this.salary+"]";
  }
  ...
}

public class Manager extends Employee {
  ...
  public String toString() {
    return super.toString()+"[bonus="+this.bonus+"]";
  }
  ...
}
```

Method equals

Method `equals` tests whether one object is considered equal to another.

Method equals

Method `equals` tests whether one object is considered equal to another. The implementation in class `Object` just checks if two object references are identical.

Method equals

Method equals tests whether one object is considered equal to another. The implementation in class Object just checks if two object references are identical.

Example:

```
public class Item {
  private String description;
  private double price;
  ...
  public boolean equals( Object other ) {
    if (this == other) return true;
    if (other == null) return false;
    if (getClass() != other.getClass()) return false;
    Item otherItem = (Item) other;
    return
      Objects.equals(this.description,otherItem.description)
      &&(this.price == other.price);
  }
}
```

Method equals (2)

```java
public class DiscountedItem extends Item {
  private double discount;
  ...
  public boolean equals( Object other ) {
    if (!super.equals(other)) return false;
    DiscountedItem otherItem = (DiscountedItem) other;
    return this.discount == otherItem.discount;
  }
}
```

Method hashCode

A hash code is an integer that is derived from an object.

Method hashCode

A hash code is an integer that is derived from an object.

Hash codes should be scrambled, if `x` and `y` are two unequal objects, `x.hashCode()` and `y.hashCode()` should be different with high probability.

Method hashCode

A hash code is an integer that is derived from an object.

Hash codes should be scrambled, if x and y are two unequal objects, x.hashCode() and y.hashCode() should be different with high probability.

Hash code algorithm for String:

```
int hash = 0;
for( int i=0; i<length(); i++) {
  hash = 31*hash + charAt(i);
}
```

Method hashCode

A hash code is an integer that is derived from an object.

Hash codes should be scrambled, if x and y are two unequal objects, x.hashCode() and y.hashCode() should be different with high probability.

Hash code algorithm for String:

```
int hash = 0;
for( int i=0; i<length(); i++) {
  hash = 31*hash + charAt(i);
}
```

Util method in class Objects:

```
public int hashCode() {
  return Objects.hash(description,price);
}
```

hashCode contract

The general contract of hashCode is:

hashCode contract

The general contract of hashCode is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer.

hashCode contract

The general contract of hashCode is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer.
- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

Equals contract

The equals method implements an equivalence relation on non-null object references:

Equals contract

The equals method implements an equivalence relation on non-null object references:

- It is reflexive: for any non-null reference value `x`, `x.equals(x)` should return `true`.

Equals contract

The equals method implements an equivalence relation on non-null object references:

- It is reflexive: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is symmetric: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.

Equals contract

The equals method implements an equivalence relation on non-null object references:

- It is reflexive: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is symmetric: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is transitive: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.

Equals contract

The equals method implements an equivalence relation on non-null object references:

- It is reflexive: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is symmetric: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is transitive: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is consistent: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in equals comparisons on the objects is modified.

Equals contract

The equals method implements an equivalence relation on non-null object references:

- It is reflexive: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is symmetric: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is transitive: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is consistent: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

Cloning Objects

Method clone is used to make a clone.

Cloning Objects

Method clone is used to make a clone. This method is declared protected so we can override it if needed.

Cloning Objects

Method clone is used to make a clone. This method is declared protected so we can override it if needed.

The default implementation performs a shallow copy.

Cloning Objects

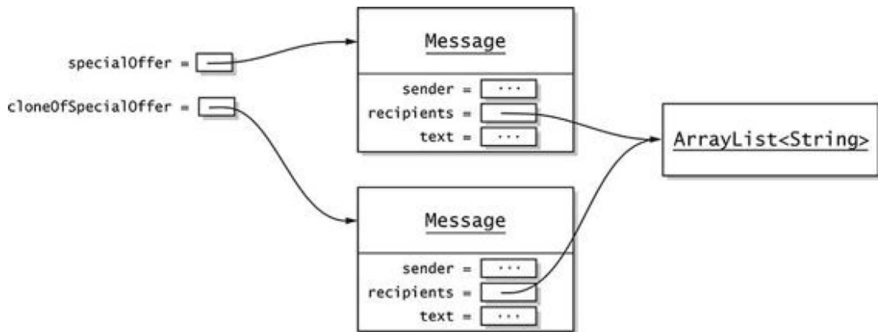
Method clone is used to make a clone. This method is declared protected so we can override it if needed.

The default implementation performs a shallow copy.

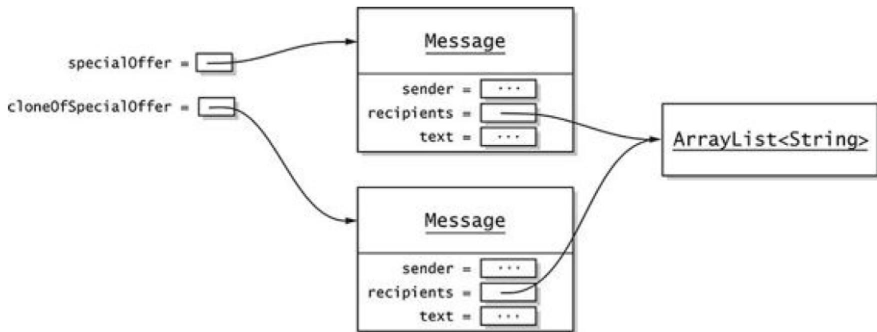
It only works for basic types:

```
public final class Message {
  private String sender;
  private ArrayList<String> recipients;
  private Strin text;
  ...
  public void addRecipient( String recipient ) { ... }
}
```

Cloning Objects



Cloning Objects



A deep copy is needed!

Cloning Objects

When we implement a class, we have to decide whether:

Cloning Objects

When we implement a class, we have to decide whether:

1. We do not want to provide a `clone` method

Cloning Objects

When we implement a class, we have to decide whether:

1. We do not want to provide a `clone` method
2. The inherited `clone` method is acceptable

Cloning Objects

UNICAM
Università di Camerino
1336

When we implement a class, we have to decide whether:

1. We do not want to provide a clone method
2. The inherited clone method is acceptable
3. The clone method should make a deep copy

do nothing!

Cloning Objects

When we implement a class, we have to decide whether:

1. We do not want to provide a `clone` method

   do nothing!

2. The inherited `clone` method is acceptable

   Implement interface `Cloneable`!

3. The `clone` method should make a deep copy

   Override method `clone`!

Enumerations

An enum type is a special data type that enables for a variable to be a set of predefined constants:

Enumerations

An enum type is a special data type that enables for a variable to be a set of predefined constants:

```
public enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE; }
```

Enumerations

An enum type is a special data type that enables for a variable to be a set of predefined constants:

```
public enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE; }
```

Elements of an enumeration can be compared with the == operator (there is only one instance of each case).

Enumerations

An enum type is a special data type that enables for a variable to be a set of predefined constants:

```
public enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE; }
```

Elements of an enumeration can be compared with the == operator (there is only one instance of each case).

Method `toString` by default provides the name of the enumerated object (e.g. "SMALL").

Enumerations

The method valueOf can be used to build an element of the enumeration from string:

```
Size notMySize = Size.valueOf("SMALL");
```

Enumerations

The method valueOf can be used to build an element of the enumeration from string:

```
Size notMySize = Size.valueOf("SMALL");
```

Each enumerated type has a static method values that returns an array of all instances:

```
Size[] allValues = Size.values();
```

Enumerations

The method valueOf can be used to build an element of the enumeration from string:

```
Size notMySize = Size.valueOf("SMALL");
```

Each enumerated type has a static method values that returns an array of all instances:

```
Size[] allValues = Size.values();
```

Method original can be used to get the position of an instance in the enum declaration.

Enumerations

The method valueOf can be used to build an element of the enumeration from string:

```
Size notMySize = Size.valueOf("SMALL");
```

Each enumerated type has a static method values that returns an array of all instances:

```
Size[] allValues = Size.values();
```

Method original can be used to get the position of an instance in the enum declaration.

Any enumerate type E implements Comparable<E>, the comparison is performed via original values.

Constructors, Methods, and Fields

If needed we can add constructors, methods, and fields to an enumeration type:

Constructors, Methods, and Fields

If needed we can add constructors, methods, and fields to an enumeration type:

```
public enum Size {
  SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

   private String abbreviation;

   Size(String abbreviation) {
     this.abbreviation = abbreviation;
   }

   public String getAbbreviation() { return abbreviation; }

}
```

Bodies of Instances

Each `enum` instance can have specific methods.

Bodies of Instances

Each `enum` instance can have specific methods.

These have to override methods defined in the enumeration.

Bodies of Instances

Each enum instance can have specific methods.

These have to override methods defined in the enumeration.

```
public enum Opertion {
  ADD {public int eval(int arg1,int arg2) {return arg1+arg2;}

  SUB {public int eval(int arg1,int arg2) {return arg1-arg2;}

  MUL {public int eval(int arg1,int arg2) {return arg1*arg2;}

  DIV {public int eval(int arg1,int arg2) {return arg1/arg2;}

   public abstract int eval(int arg1, int arg2);
}
```

To be continued...