1. Association, Composition and Aggregation in Java

[**2.9**](https://www.geeksforgeeks.org/easy/)

**Association**

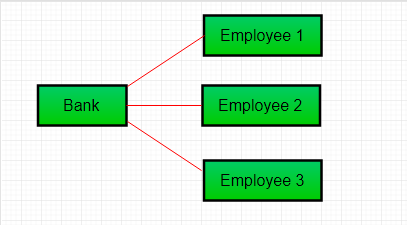
Association is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many.  
In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association.  
[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/Associatn.png)

|  |
| --- |
| // Java program to illustrate the  // concept of Association  import java.io.\*;    // class bank  class Bank  {      private String name;        // bank name      Bank(String name)      {          this.name = name;      }        public String getBankName()      {          return this.name;      }  }    // employee class  class Employee  {      private String name;        // employee name      Employee(String name)      {          this.name = name;      }        public String getEmployeeName()      {          return this.name;      }  }    // Association between both the  // classes in main method  class Association  {      public static void main (String[] args)      {          Bank bank = new Bank("Axis");          Employee emp = new Employee("Neha");            System.out.println(emp.getEmployeeName() +                 " is employee of " + bank.getBankName());      }  } |

Run on IDE

Output:

Neha is employee of Axis

In above example two separate classes Bank and Employee are associated through their Objects. Bank can have many employees, So it is a one-to-many relationship.  
[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/Aggre.png)

**Aggregation**

It is a special form of Association where:

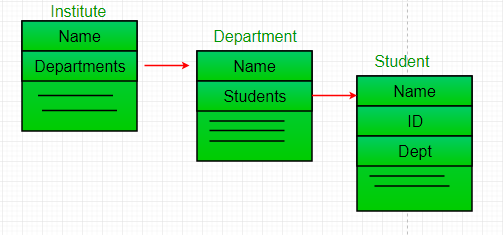
* It represents **Has-A** relationship.
* It is a **unidirectional association** i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
* In Aggregation,**both the entries can survive individually** which means ending one entity will not effect the other entity

|  |
| --- |
| // Java program to illustrate  //the concept of Aggregation.  import java.io.\*;  import java.util.\*;    // student class  class Student  {      String name;      int id ;      String dept;        Student(String name, int id, String dept)      {            this.name = name;          this.id = id;          this.dept = dept;        }  }    /\* Department class contains list of student  Objects. It is associated with student  class through its Object(s). \*/  class Department  {        String name;      private List<Student> students;      Department(String name, List<Student> students)      {            this.name = name;          this.students = students;        }        public List<Student> getStudents()      {          return students;      }  }    /\* Institute class contains list of Department  Objects. It is asoociated with Department  class through its Object(s).\*/  class Institute  {        String instituteName;      private List<Department> departments;        Institute(String instituteName, List<Department> departments)      {          this.instituteName = instituteName;          this.departments = departments;      }        // count total students of all departments      // in a given institute      public int getTotalStudentsInInstitute()      {          int noOfStudents = 0;          List<Student> students;          for(Department dept : departments)          {              students = dept.getStudents();              for(Student s : students)              {                  noOfStudents++;              }          }          return noOfStudents;      }    }    // main method  class GFG  {      public static void main (String[] args)      {          Student s1 = new Student("Mia", 1, "CSE");          Student s2 = new Student("Priya", 2, "CSE");          Student s3 = new Student("John", 1, "EE");          Student s4 = new Student("Rahul", 2, "EE");            // making a List of          // CSE Students.          List <Student> cse\_students = new ArrayList<Student>();          cse\_students.add(s1);          cse\_students.add(s2);            // making a List of          // EE Students          List <Student> ee\_students = new ArrayList<Student>();          ee\_students.add(s3);          ee\_students.add(s4);            Department CSE = new Department("CSE", cse\_students);          Department EE = new Department("EE", ee\_students);            List <Department> departments = new ArrayList<Department>();          departments.add(CSE);          departments.add(EE);            // creating an instance of Institute.          Institute institute = new Institute("BITS", departments);            System.out.print("Total students in institute: ");          System.out.print(institute.getTotalStudentsInInstitute());      }  } |

Run on IDE

Output:

Total students in institute: 4

In this example, there is an Institute which has no. of departments like CSE, EE. Every department has no. of students. So, we make a Institute class which has a reference to Object or no. of Objects (i.e. List of Objects) of the Department class. That means Institute class is associated with Department class through its Object(s). And Department class has also a reference to Object or Objects (i.e. List of Objects) of Student class means it is associated with Student class through its Object(s).  
It represents a **Has-A** relationship.  
[](https://www.geeksforgeeks.org/cdncontribute.geeksforgeeks.org/wp-content/uploads/Reference.png)

**When do we use Aggregation ??**  
Code reuse is best achieved by aggregation.

**Composition**

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

* It represents **part-of** relationship.
* In composition, both the entities are dependent on each other.
* When there is a composition between two entities, the composed object **cannot exist**without the other entity.

Lets take example of**Library**.

|  |
| --- |
| // Java program to illustrate  // the concept of Composition  import java.io.\*;  import java.util.\*;    // class book  class Book  {        public String title;      public String author;        Book(String title, String author)      {            this.title = title;          this.author = author;      }  }    // Libary class contains  // list of books.  class Library  {        // reference to refer to list of books.      private final List<Book> books;        Library (List<Book> books)      {          this.books = books;      }        public List<Book> getTotalBooksInLibrary(){           return books;      }    }    // main method  class GFG  {      public static void main (String[] args)      {            // Creating the Objects of Book class.          Book b1 = new Book("EffectiveJ Java", "Joshua Bloch");          Book b2 = new Book("Thinking in Java", "Bruce Eckel");          Book b3 = new Book("Java: The Complete Reference", "Herbert Schildt");            // Creating the list which contains the          // no. of books.          List<Book> books = new ArrayList<Book>();          books.add(b1);          books.add(b2);          books.add(b3);            Library library = new Library(books);            List<Book> bks = library.getTotalBooksInLibrary();          for(Book bk : bks){                System.out.println("Title : " + bk.title + " and "              +" Author : " + bk.author);          }      }  } |

Run on IDE

Output

Title : EffectiveJ Java and Author : Joshua Bloch

Title : Thinking in Java and Author : Bruce Eckel

Title : Java: The Complete Reference and Author : Herbert Schildt

In above example a library can have no. of **books** on same or different subjects. So, If Library gets destroyed then All books within that particular library will be destroyed. i.e. book can not exist without library. That’s why it is composition.

**Aggregation vs Composition**

1. **Dependency:** Aggregation implies a relationship where the child **can exist independently**of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child **cannot exist independent** of the parent. Example: Human and heart, heart don’t exist separate to a Human
2. **Type of Relationship:** Aggregation relation is **“has-a”** and composition is **“part-of”**relation.
3. **Type of association:**Composition is a **strong** Association whereas Aggregation is a **weak**Association.

|  |
| --- |
| // Java program to illustrate the  // difference between Aggregation  // Composition.    import java.io.\*;    // Engine class which will  // be used by car. so 'Car'  // class will have a field  // of Engine type.  class Engine  {      // starting an engine.      public void work()      {            System.out.println("Engine of car has been started ");        }    }    // Engine class  final class Car  {        // For a car to move,      // it need to have a engine.      private final Engine engine; // Composition      //private Engine engine;     // Aggregation        Car(Engine engine)      {          this.engine = engine;      }        // car start moving by starting engine      public void move()      {            //if(engine != null)          {              engine.work();              System.out.println("Car is moving ");          }      }  }    class GFG  {      public static void main (String[] args)      {            // making an engine by creating          // an instance of Engine class.          Engine engine = new Engine();            // Making a car with engine.          // so we are passing a engine          // instance as an argument while          // creating instace of Car.          Car car = new Car(engine);          car.move();        }  } |

Run on IDE

Output:

Engine of car has been started

Car is moving

# JDK 7 Features:

## Diamond Operator

## Using strings in switch statements

## Java 7 JDBC Improvements Automatic resource management

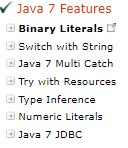
## Numeric literals with underscores

## Improved exception handling

## New file system API (NIO 2.0)

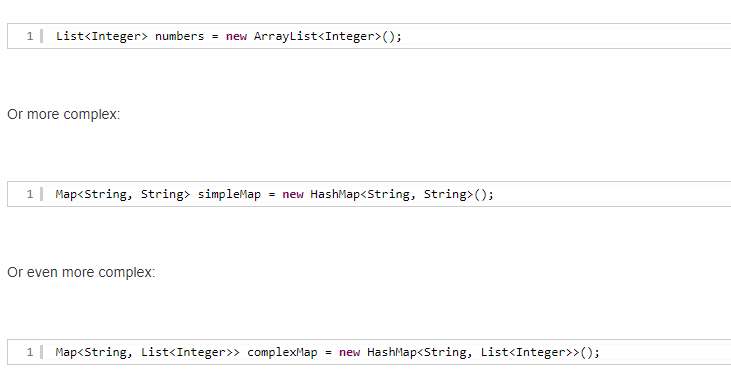
## Fork and Join

## Supporting dynamism

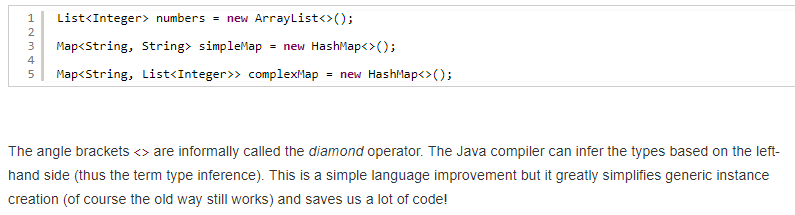


## Diamond Operator Examples

Old style of list,map,set declarations.

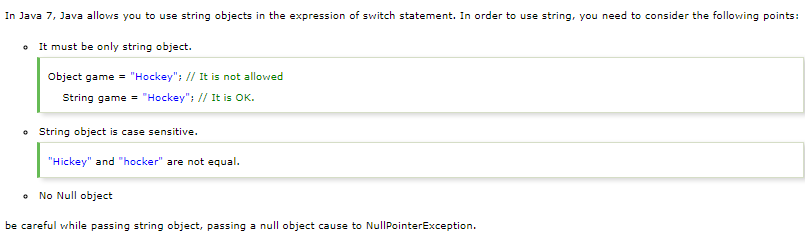


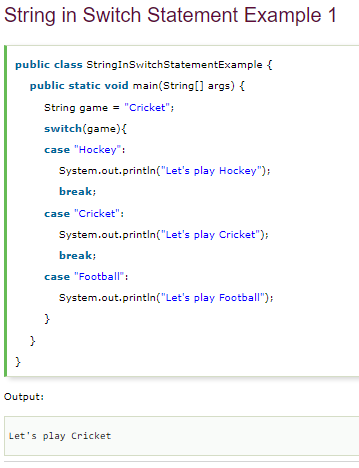
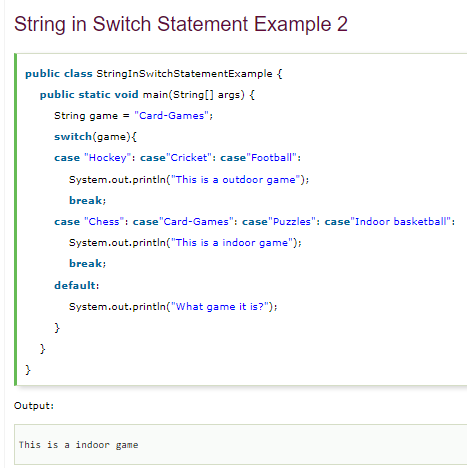
Why do we have to declare the parameter types on both side? The parameter types on the right-hand side seem to be redundant, right? So in Java 7 and later, developer can now replace the right-hand side parameters with an empty set of parameters like this:



## Using strings in switch statements Examples

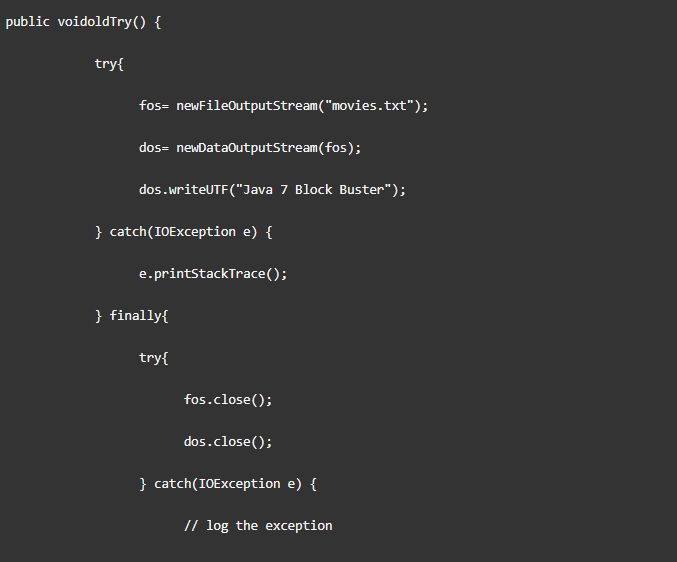
String in Switch Statement



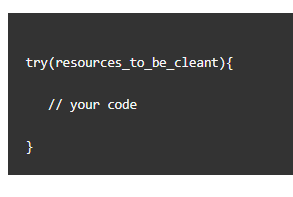
 

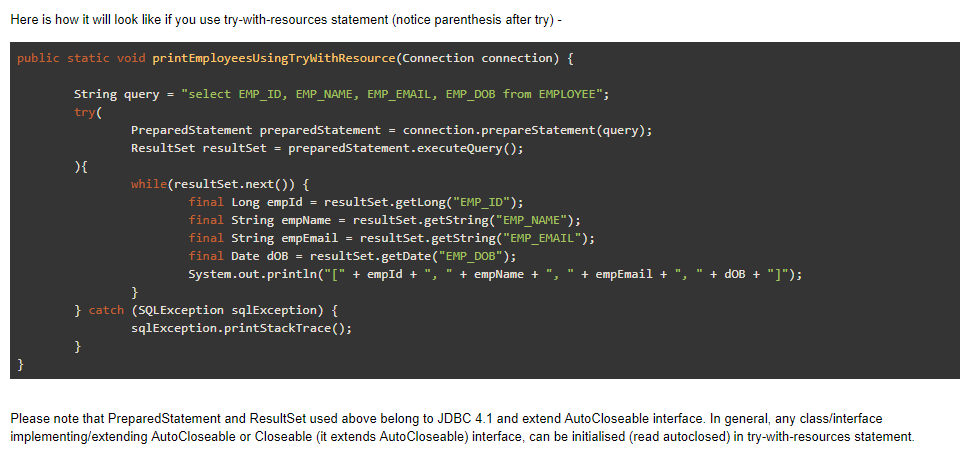
## Java 7 JDBC Improvements -Automatic resource management

Resources such as Connections, Files, Input/OutStreams, etc. should be closed manually by the developer by writing bog-standard code. Usually we use a try-finally block to close the respective resources. See the current practice of creating a resource, using it and finally closing it:

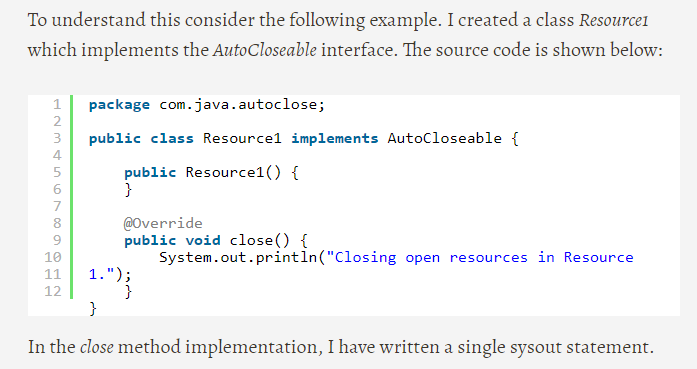


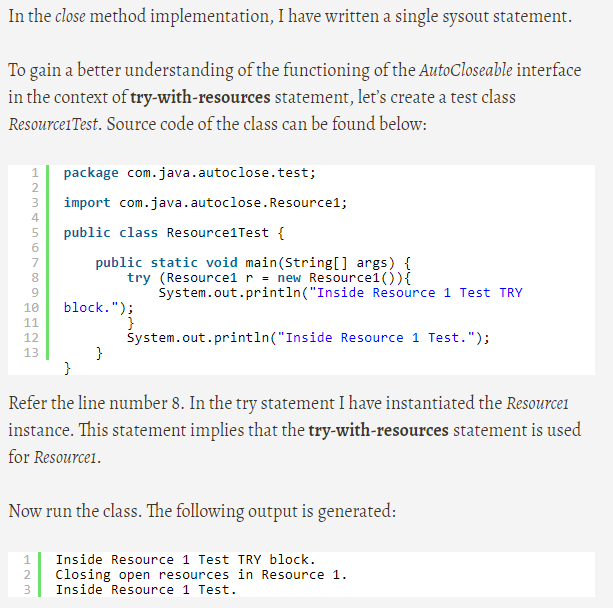
However, Java 7 has introduced another cool feature to manage the resources automatically. It is simple in operation, too. All we have to do is declare the resources in the try as follows:

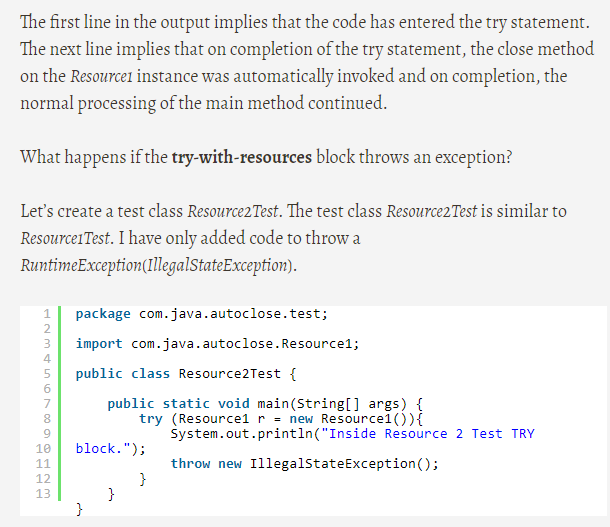
 

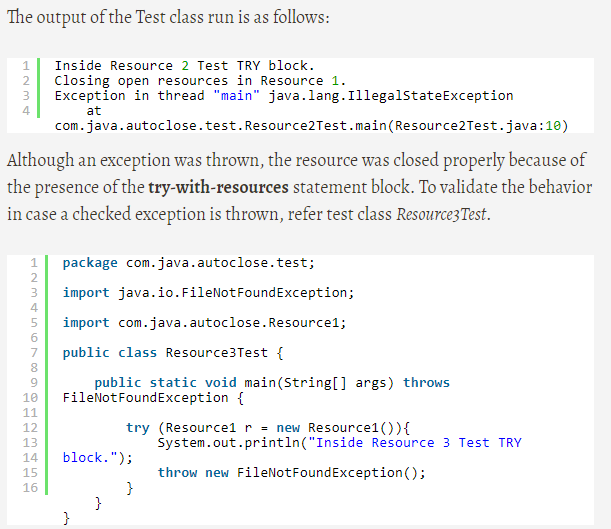


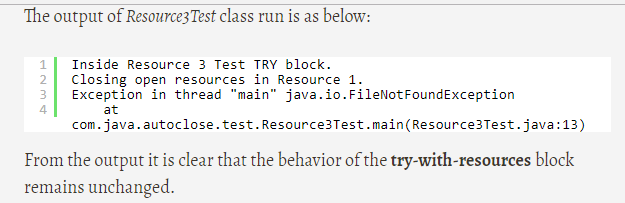
During our day-to-day programming, we make use of number of resources which need to have their **close** method invoked on completion of their use in our code. The common example in I/O is [java.io.FileInputStream](http://docs.oracle.com/javase/7/docs/api/java/io/FileInputStream.html" \t "_blank" \o "java.io.FileInputStream JavaDoc) or in JDBC is [java.sql.Connection](http://docs.oracle.com/javase/7/docs/api/java/sql/Connection.html" \t "_blank" \o "java.sql.Connection JavaDoc). These classes maintain a handle to the physical File or database connection which needs to be released as soon as its use is completed so that other processes can utilize the same. Java 7 has introduced the new interface **AutoCloseable** which defines a single close method. The interface implementations of **AutoCloseable** will in conjunction with the newly introduced **try-with-resources** statement provide a methodology to ensure that the resource is closed as soon as its operations are completed.





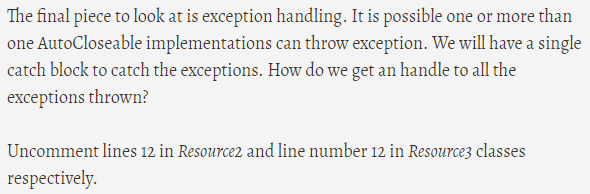


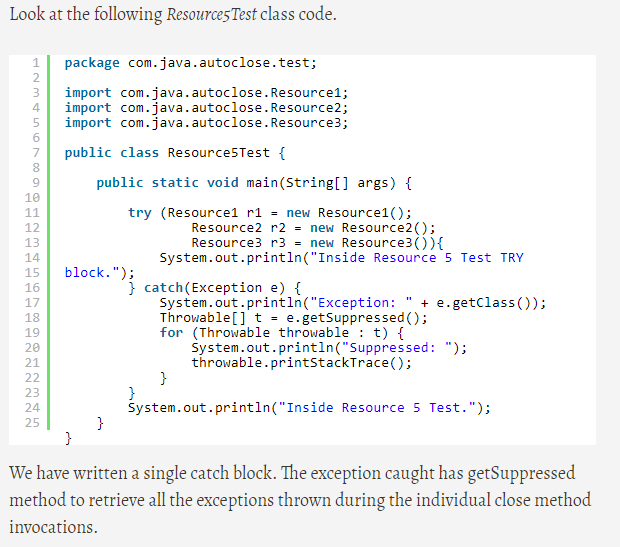


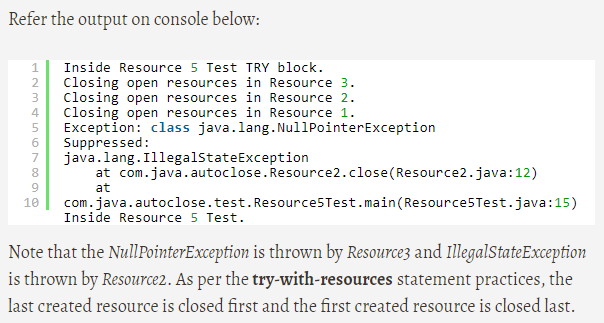














As you can observe, we have skipped the close() method of the [BufferedReader](http://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html) class, due to the fact that it is called automatically by the try-with-resources statement.

A sample execution is shown below:

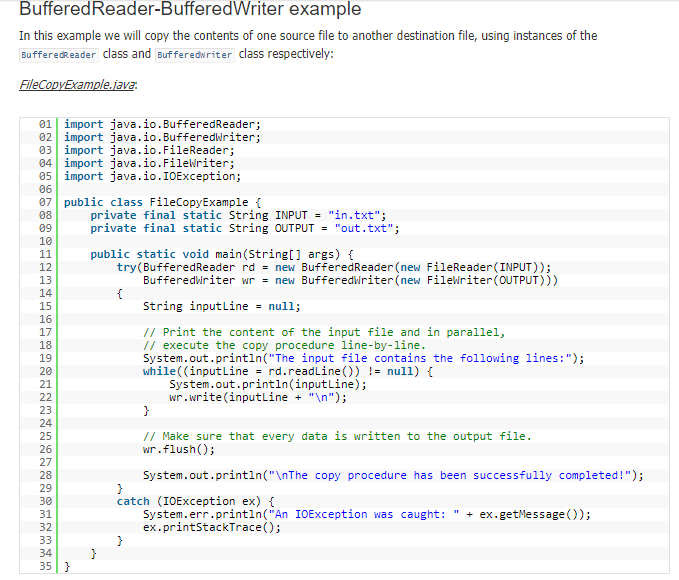
|  |  |
| --- | --- |
| 1 | Hello from |
| 2 | Java Code Geeks! |

In versions before Java 1.7, a resource was closed inside the finally statement, in order to ensure that the resource was successfully closed, regardless of whether the try statement completes normally or not. Thus, our previous example would be transformed to:



### The AutoCloseable Interface in Java

A large number of classes implements the [AutoCloseable](http://docs.oracle.com/javase/8/docs/api/java/lang/AutoCloseable.html) interface. In this section, we will describe some of them and we will demonstrate examples where multiple resources are declared inside a try-with-resources statement. Every class that implements the [AutoCloseable](http://docs.oracle.com/javase/8/docs/api/java/lang/AutoCloseable.html) interface must define the behaviour of the [close()](http://docs.oracle.com/javase/8/docs/api/java/lang/AutoCloseable.html#close--) method.



Notice that multiple resource definitions are separated by the ; character.

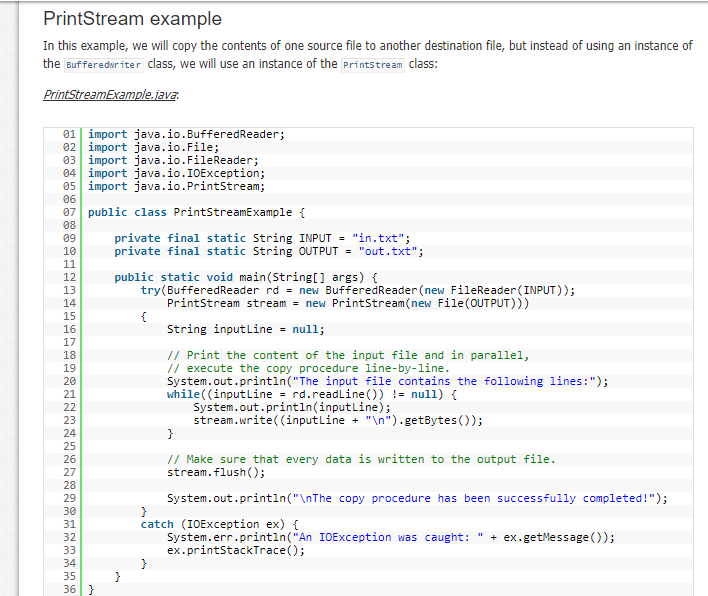
A sample execution is shown below:

[view source](https://examples.javacodegeeks.com/core-java/java-autocloseable-interface-example/#viewSource)[print](https://examples.javacodegeeks.com/core-java/java-autocloseable-interface-example/#printSource)[?](https://examples.javacodegeeks.com/core-java/java-autocloseable-interface-example/#about)

|  |  |
| --- | --- |
| 1 | The input file contains the following lines: |
| 2 | Hello from |

|  |  |
| --- | --- |
| 3 | Java Code Geeks! |
| 4 |  |

|  |  |
| --- | --- |
| 5 | The copy procedure has been successfully completed! |



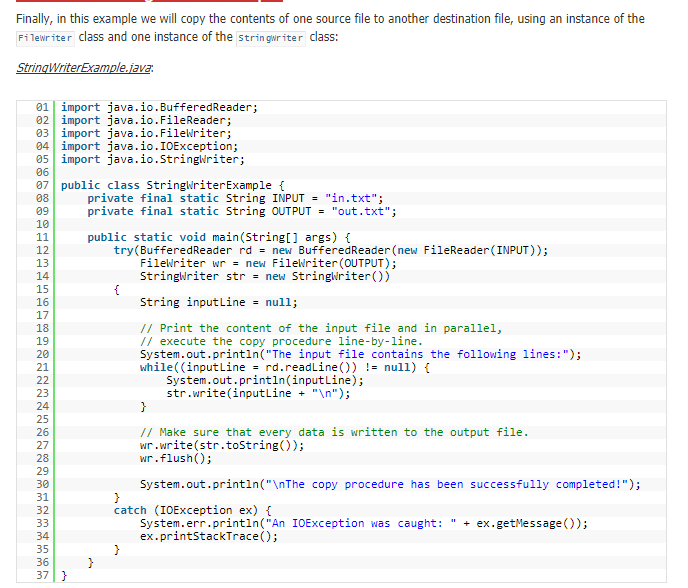
A sample execution is shown below:

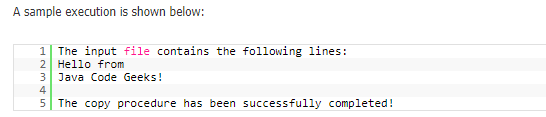
|  |  |
| --- | --- |
| 1 | The input file contains the following lines: |
| 2 | Hello from |

|  |  |
| --- | --- |
| 3 | Java Code Geeks! |
| 4 |  |

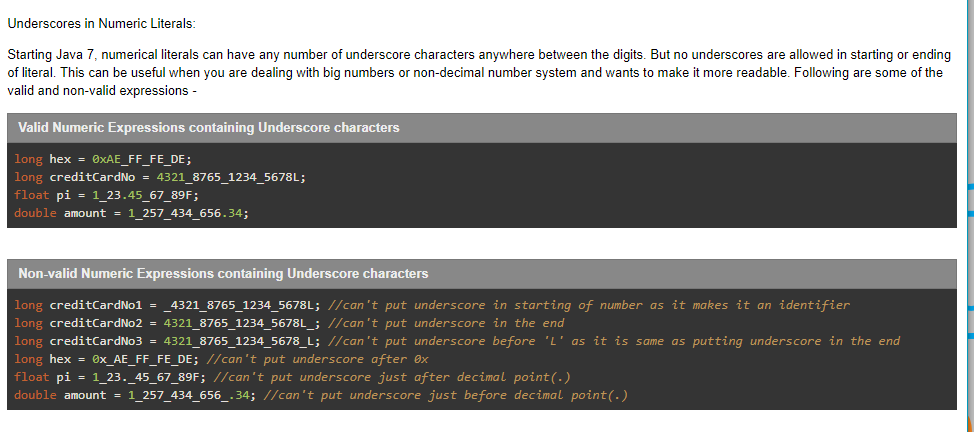
|  |  |
| --- | --- |
| 5 | The copy procedure has been successfully completed! |

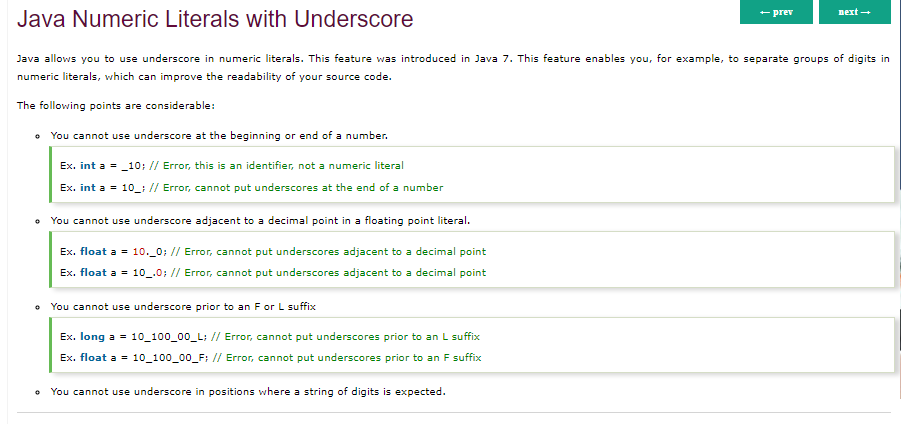
### FileWriter-StringWriter example

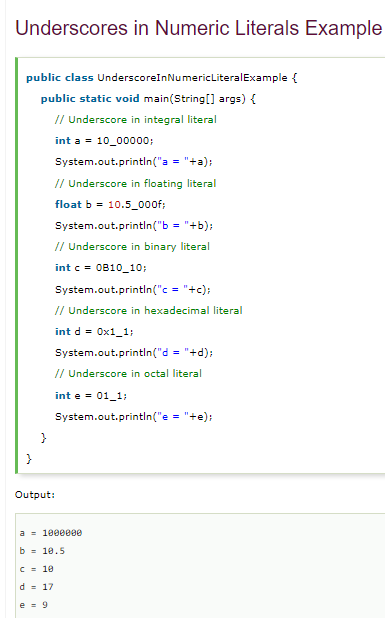




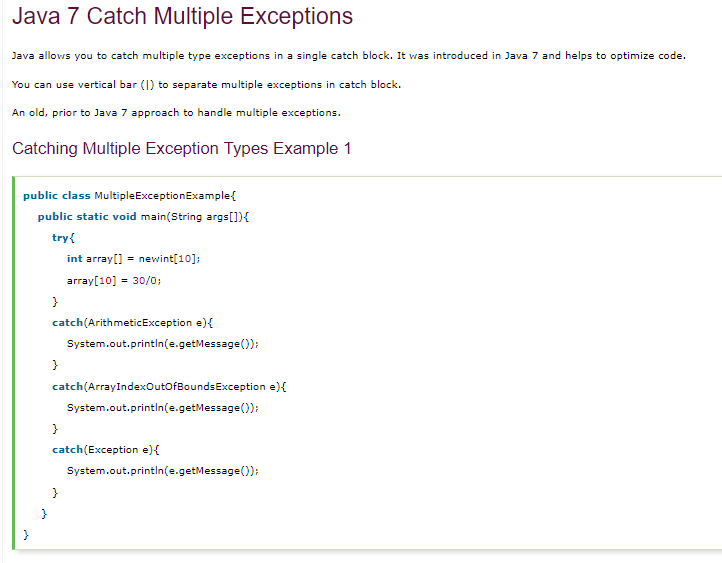
## Numeric literals with underscores

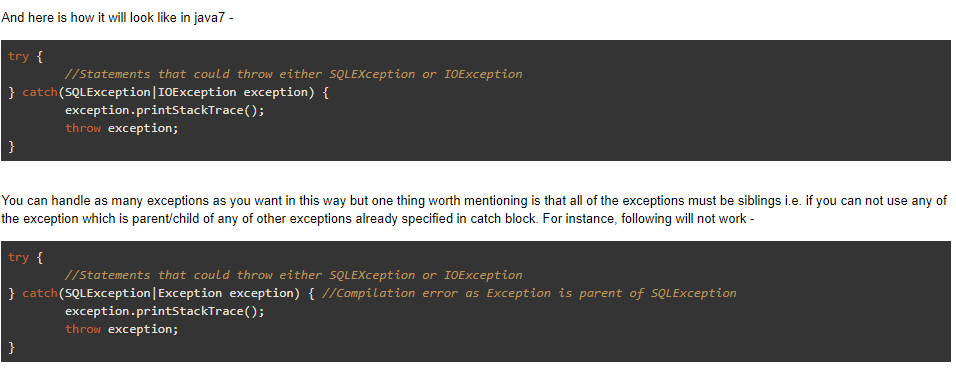




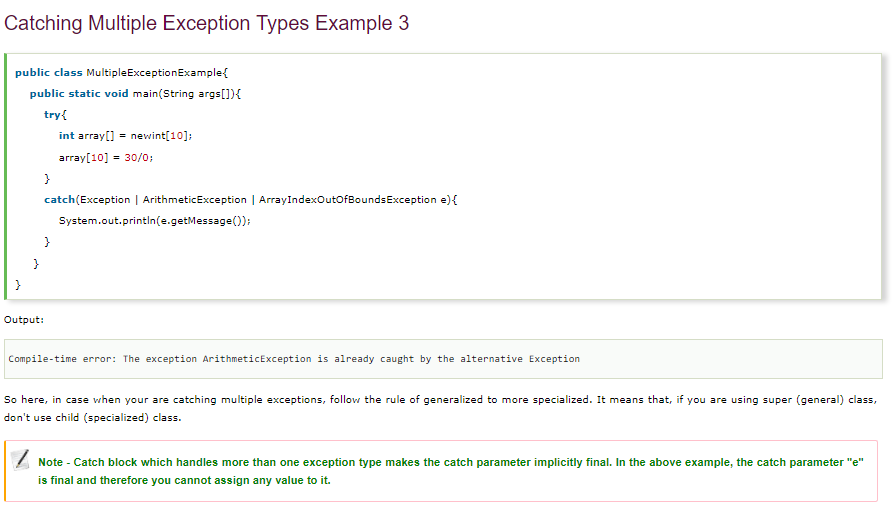


## Improved exception handling









## New file system API (NIO 2.0)

## Fork and Join

## Supporting dynamism

# immutability in Java

In order to get the benefits of immutability in Java we can use the following guidelines while programming -

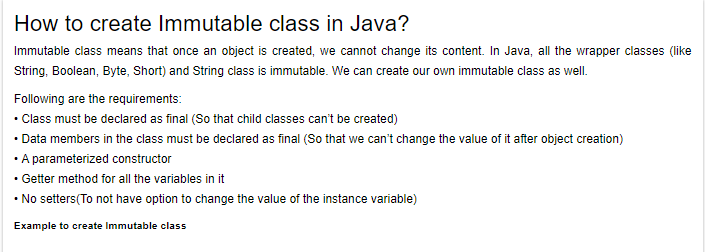
* Mark the class **final**
* Mark all the fields **private** and **final**
* Force all the callers to construct an object of the class directly, i.e. do not use any **setter** methods
* Do not change the state of the objects in any methods of the class

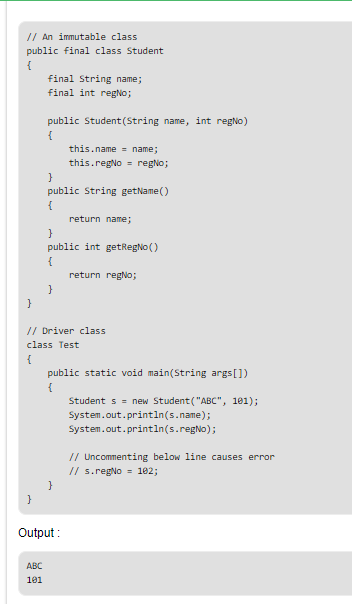
Doing so will often restrict the way you can call the class and its methods. It will also make you redesign your software and rethink your algorithms. However, there are many benefits of programming with immutable objects.

1. Immutable objects are thread-safe so you will not have any synchronization issues.
2. Immutable objects are good **Map** keys and **Set** elements, since these typically do not change once created.
3. Immutability makes it easier to write, use and reason about the code (class invariant is established once and then unchanged)
4. Immutability makes it easier to parallelize your program as there are no conflicts among objects.
5. The internal state of your program will be consistent even if you have exceptions.
6. References to immutable objects can be cached as they are not going to change.

As a good programming practice in Java one should try to use immutable objects as far as possible. Immutability can have a performance cost, since when an object cannot be mutated we need to copy it if we want to write to it. When you care a lot about performance (e.g. programming a game) it may be necessary to use a mutable object. Even then it is often better to try to limit the mutability of objects. This recommendation can be summarized in the following adage by Joshua Bloch (taken from the book [Effective Java](http://amzn.to/1lQDbGW)) -

*Classes should be immutable unless there's a very good reason to make them mutable....If a class cannot be made immutable, limit its mutability as much as possible.*





### 2. Some popular immutable classes

***String*** is the most popular immutable class in java,  once initialized its value cannot be modified, operations like **trim(), substring(), replace()** always return a new instance and don’t affect the current instance, that’s why we usually call **trim()**as the following:

|  |  |
| --- | --- |
| 1  2 | String alex = "Alex";  alex = alex.trim(); |

Another example from JDK is the wrapper classes like: **Integer, Float, Boolean** … these classes don’t modify their state , however they create a new instance each time you try to modify them.

|  |  |
| --- | --- |
| 1  2 | Integer a =3;  a += 3; |

After calling **a += 3,** a new instance is created holding value: 6 and the first instance is lost.

### 3. How do we create an immutable class

In order to create an immutable class, you should follow the below steps:

1. Make your class **final,**so that no other classes can extend it.
2. Make all your fields **final,**so that they’re initialized only once inside the constructor and never modified afterwards.
3. Don’t expose setter methods.
4. When exposing methods which modify the state of the class, you must always return a new instance of the class.
5. If the class holds a mutable object:
   * Inside the constructor, make sure to use a clone copy of the passed argument and never set your mutable field to the real instance passed through constructor,  this is to prevent the clients who pass the object from modifying it afterwards.
   * Make sure to always return a clone copy of the field and never return the real object instance.

#### 3.1. Simple immutable class

Let’s follow the above steps and create our own immutable class (**ImmutableStudent.java**).

ImmutableStudent.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | package com.programmer.gate.beans;    public final class ImmutableStudent {        private final int id;      private final String name;        public ImmutableStudent(int id, String name) {          this.name = name;          this.id = id;      }        public int getId() {          return id;      }        public String getName() {          return name;      }  } |

The above class is a very simple immutable class which doesn’t hold any mutable object and never expose its fields in any way, these type of classes are normally used for caching purposes.

#### 3.2.  Passing mutable objects to immutable class

Now let’s complicate our example a bit, we create a mutable class called **Age**and add it as a field to **ImmutableStudent:**

Age.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | package com.programmer.gate.beans;    public class Age {        private int day;      private int month;      private int year;        public int getDay() {          return day;      }        public void setDay(int day) {          this.day = day;      }        public int getMonth() {          return month;      }        public void setMonth(int month) {          this.month = month;      }        public int getYear() {          return year;      }        public void setYear(int year) {          this.year = year;      }    } |

ImmutableStudent.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | package com.programmer.gate.beans;    public final class ImmutableStudent {        private final int id;      private final String name;      private final Age age;        public ImmutableStudent(int id, String name, Age age) {          this.name = name;          this.id = id;          this.age = age;      }        public int getId() {          return id;      }        public String getName() {          return name;      }        public Age getAge() {          return age;      }  } |

So, we added a new mutable field of type **Age** to our immutable class and assign it as normal inside the constructor.

Let’s create a simple test class and verify that **ImmutableStudent** is no more immutable:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public static void main(String[] args) {            Age age = new Age();          age.setDay(1);          age.setMonth(1);          age.setYear(1992);          ImmutableStudent student = new ImmutableStudent(1, "Alex", age);            System.out.println("Alex age year before modification = " + student.getAge().getYear());          age.setYear(1993);          System.out.println("Alex age year after modification = " + student.getAge().getYear());      } |

After running the above test, we get the following output:

|  |  |
| --- | --- |
| 1  2 | Alex age year before modification = 1992  Alex age year after modification = 1993 |

We claim that **ImmutableStudent** is an immutable class whose state is never modified after construction, however in the above example we are able to modify the age of **Alex**even after constructing **Alex** object. If we go back to the implementation of **ImmutableStudent** constructor, we find that age field is being assigned to the instance of the ***Age*** argument, so whenever the referenced **Age** is modified outside the class , the change is reflected directly on the state of **Alex.**Check [Pass by value OR pass by reference article](http://programmergate.com/java-pass-reference-pass-value/) to deeply understand this concept.

In order to fix this and make our class again immutable, we follow step **#5** from the steps that we mention above for creating an immutable class. So we modify the constructor in order to clone the passed argument of **Age** and use a clone instance of it.

ImmutableStudent.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public ImmutableStudent(int id, String name, Age age) {          this.name = name;          this.id = id;          Age cloneAge = new Age();          cloneAge.setDay(age.getDay());          cloneAge.setMonth(age.getMonth());          cloneAge.setYear(age.getYear());          this.age = cloneAge;      } |

Now, if we run our test, we get the following output:

|  |  |
| --- | --- |
| 1  2 | Alex age year before modification = 1992  Alex age year after modification = 1992 |

As you see now, the age of **Alex** is never affected after construction and our class is back immutable.

#### 3.3. Returning mutable objects from immutable class

However, our class still have a leak and is not fully immutable, let’s take the following test scenario:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public static void main(String[] args) {            Age age = new Age();          age.setDay(1);          age.setMonth(1);          age.setYear(1992);          ImmutableStudent student = new ImmutableStudent(1, "Alex", age);            System.out.println("Alex age year before modification = " + student.getAge().getYear());          student.getAge().setYear(1993);          System.out.println("Alex age year after modification = " + student.getAge().getYear());      } |

Output:

|  |  |
| --- | --- |
| 1  2 | Alex age year before modification = 1992  Alex age year after modification = 1993 |

Again according to step **#4**, when returning mutable fields from immutable object, you should return a clone instance of them and not the real instance of the field.

So we modify **getAge()**in order to return a clone of the object’s age:

ImmutableStudent.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public Age getAge() {          Age cloneAge = new Age();          cloneAge.setDay(this.age.getDay());          cloneAge.setMonth(this.age.getMonth());          cloneAge.setYear(this.age.getYear());            return cloneAge;      } |

Now the class becomes fully immutable and provides no way or method for other objects to modify its state.

|  |  |
| --- | --- |
| 1  2 | Alex age year before modification = 1992  Alex age year after modification = 1992 |

4. Conclusion

Immutable classes provide a lot of advantages especially when used correctly in multi-threaded environment, the only disadvantage is that they consume more memory than the traditional class since upon each modification of them a new object is created in the memory, but a developer should not overestimate the memory consumption as its negligible compared to the advantages provided by these type of classes.

Finally, an object is immutable if it can present only one state to the other objects, no matter how and when they call its methods. If so it’s thread safe by any definition of thread-safe.

# Eclipse Vesions and supported Jdk version

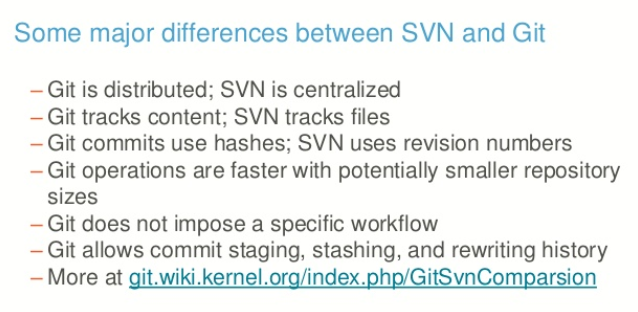
| **Version name** | **Date** | **Platform version** | **Projects** | **Main changes** |
| --- | --- | --- | --- | --- |
| [N/A](https://en.wikipedia.org/wiki/N/a) | 21 June 2004 | 3.0[[22]](https://en.wikipedia.org/wiki/Eclipse_(software)#cite_note-22) |  |  |
| N/A | 28 June 2005 | 3.1 |  | Added [Java 5](https://en.wikipedia.org/wiki/Java_5) support: generics, annotations, boxing-unboxing, enums, enhanced for loop, varargs, static imports[[23]](https://en.wikipedia.org/wiki/Eclipse_(software)#cite_note-23) |
| [Callisto](https://en.wikipedia.org/wiki/Callisto_(moon)) | 30 June 2006 | 3.2 | Callisto projects[[24]](https://en.wikipedia.org/wiki/Eclipse_(software)#cite_note-24) |  |
| [Europa](https://en.wikipedia.org/wiki/Europa_(moon)) | 29 June 2007 | 3.3 | Europa projects[[21]](https://en.wikipedia.org/wiki/Eclipse_(software)#cite_note-:0-21) |  |
| [Ganymede](https://en.wikipedia.org/wiki/Ganymede_(moon)) | 25 June 2008 | 3.4 | Ganymede projects[[25]](https://en.wikipedia.org/wiki/Eclipse_(software)#cite_note-25) |  |
| [Galileo](https://en.wikipedia.org/wiki/Galileo_Galilei) | 24 June 2009 | 3.5 | Galileo projects[[26]](https://en.wikipedia.org/wiki/Eclipse_(software)#cite_note-26) |  |
| [Helios](https://en.wikipedia.org/wiki/Helios) | 23 June 2010 | 3.6 | Helios projects[[27]](https://en.wikipedia.org/wiki/Eclipse_(software)#cite_note-27) |  |
| [Indigo](https://en.wikipedia.org/wiki/Indigo) | 22 June 2011 | 3.7 | Indigo projects[[28]](https://en.wikipedia.org/wiki/Eclipse_(software)#cite_note-28) | Added [Java 7](https://en.wikipedia.org/wiki/Java_7) support (3.7.1 sr1): Improved Type Inference for Generic Instance Creation (Diamond), Multi-catch, try-with-resources statement, Simplified Varargs Method Invocation, Strings in switch, Binary Literals and Underscores in Numeric Literals, Polymorphic Methods[[29]](https://en.wikipedia.org/wiki/Eclipse_(software)#cite_note-29) |
| [Juno](https://en.wikipedia.org/wiki/Juno_(mythology)) | 27 June 2012 | 3.8 and 4.2[[30]](https://en.wikipedia.org/wiki/Eclipse_(software)#cite_note-30)  [[Notes 1]](https://en.wikipedia.org/wiki/Eclipse_(software)#cite_note-33) | Juno projects[[33]](https://en.wikipedia.org/wiki/Eclipse_(software)#cite_note-34) |  |
| [Kepler](https://en.wikipedia.org/wiki/Johannes_Kepler) | 26 June 2013 | 4.3 | Kepler projects[[34]](https://en.wikipedia.org/wiki/Eclipse_(software)#cite_note-35) |  |
| [Luna](https://en.wikipedia.org/wiki/Moon) | 25 June 2014 | 4.4 | Luna projects[[35]](https://en.wikipedia.org/wiki/Eclipse_(software)#cite_note-36) | Integrated [Java 8](https://en.wikipedia.org/wiki/Java_8) support;[[36]](https://en.wikipedia.org/wiki/Eclipse_(software)#cite_note-37) in the prior version, this was possible via a *Java 8 patch* plug-in |
| [Mars](https://en.wikipedia.org/wiki/Mars) | 24 June 2015 | 4.5 | Mars projects[[37]](https://en.wikipedia.org/wiki/Eclipse_(software)#cite_note-38) |  |
| [Neon](https://en.wikipedia.org/wiki/Neon) | 22 June 2016 | 4.6 | Neon projects[[38]](https://en.wikipedia.org/wiki/Eclipse_(software)#cite_note-39) |  |
| [Oxygen](https://en.wikipedia.org/wiki/Oxygen) | 28 June 2017 | **4.7** | Oxygen projects[[39]](https://en.wikipedia.org/wiki/Eclipse_(software)#cite_note-40) |  |
| [Photon](https://en.wikipedia.org/wiki/Photon) | 27 June 2018 (planned) | 4.8 | Photon projects[[40]](https://en.wikipedia.org/wiki/Eclipse_(software)#cite_note-41) |  |

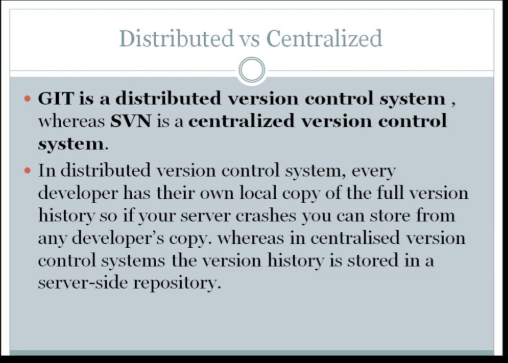
# SVN [[software versioning](https://en.wikipedia.org/wiki/Software_versioning) and [revision control](https://en.wikipedia.org/wiki/Revision_control) system]

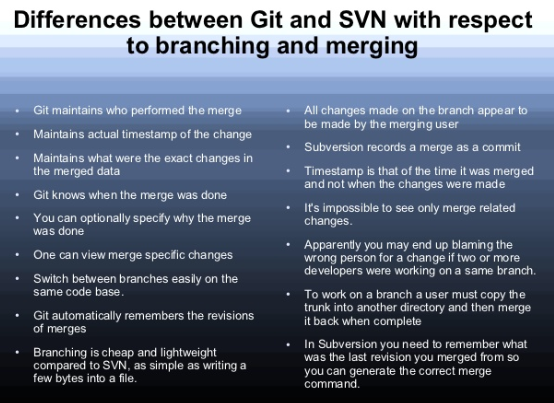
Its goal is to be a mostly compatible successor to the widely used [Concurrent Versions System](https://en.wikipedia.org/wiki/Concurrent_Versions_System) (CVS).

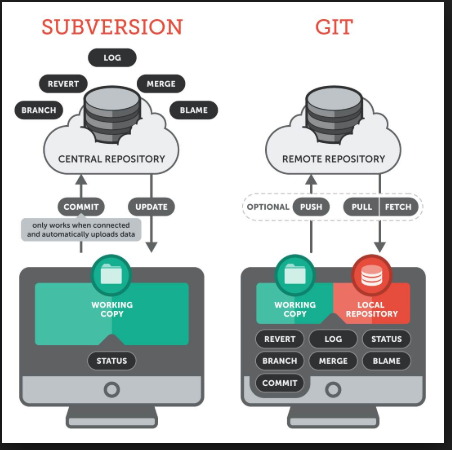


# Differences between GIT and SVN.





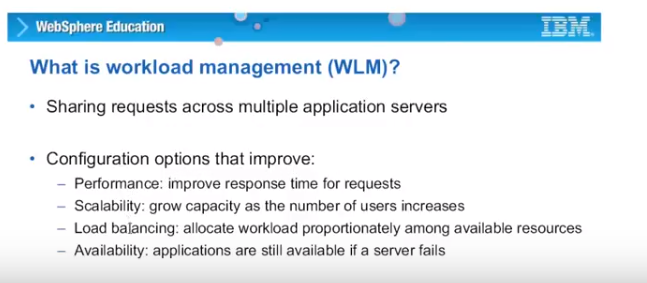


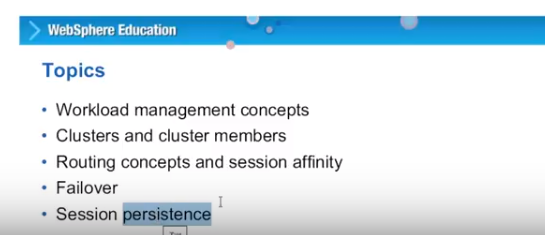


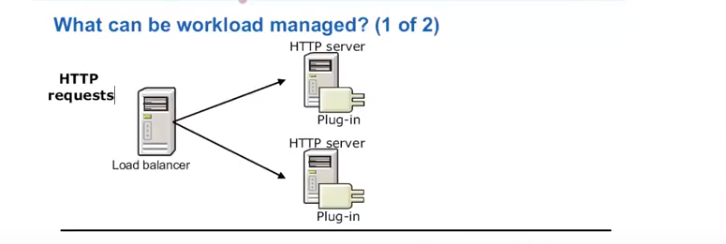
WebSphere Application Server

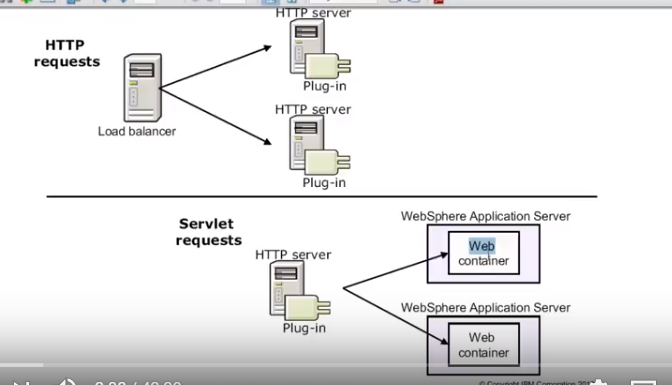


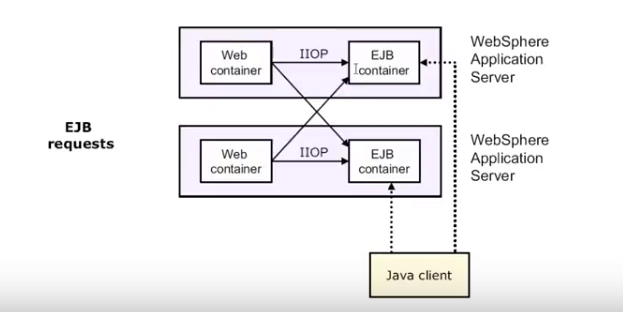
# Load Balancing in webspere:

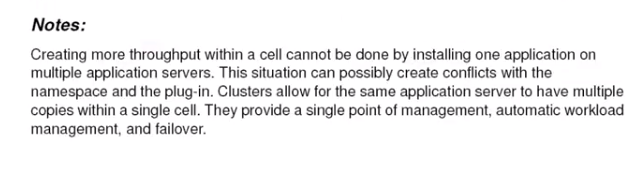


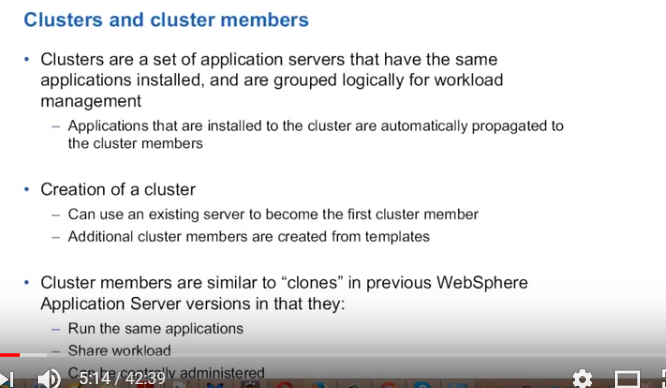


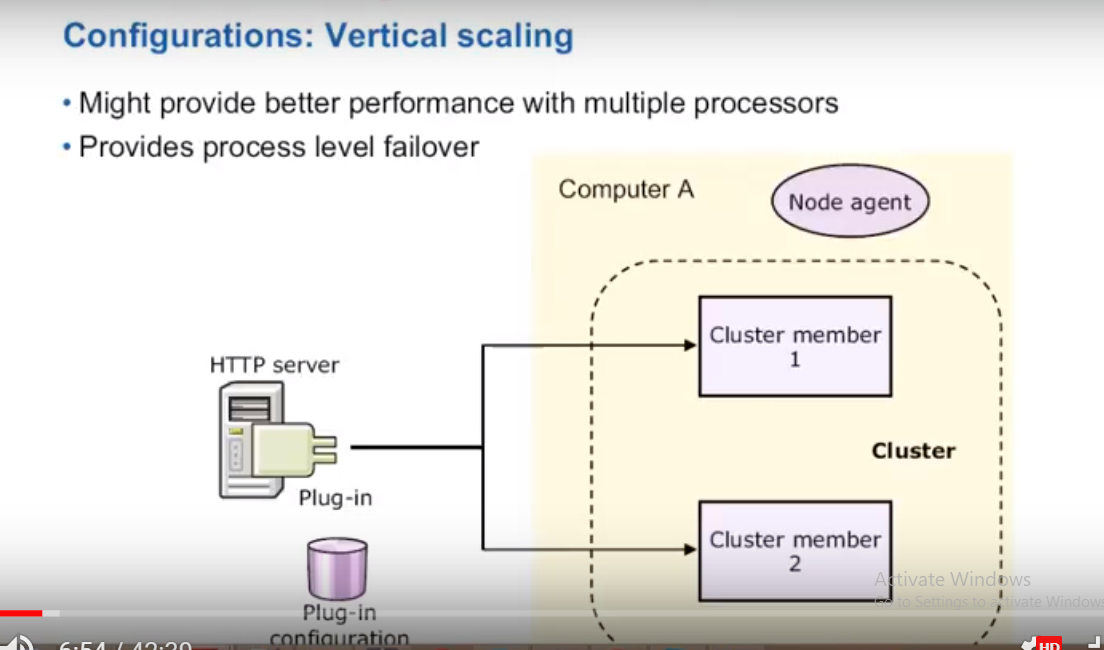


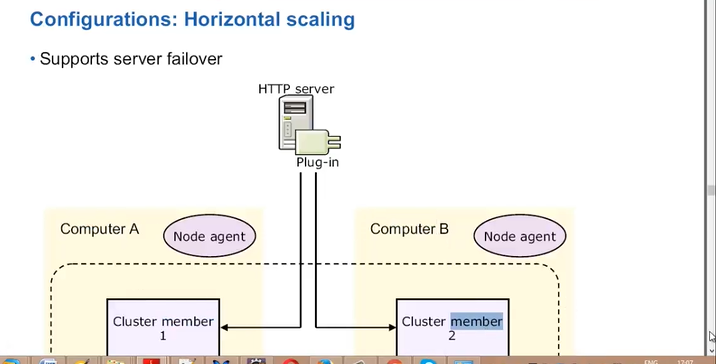


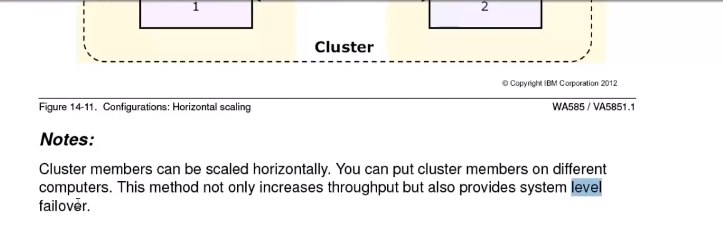


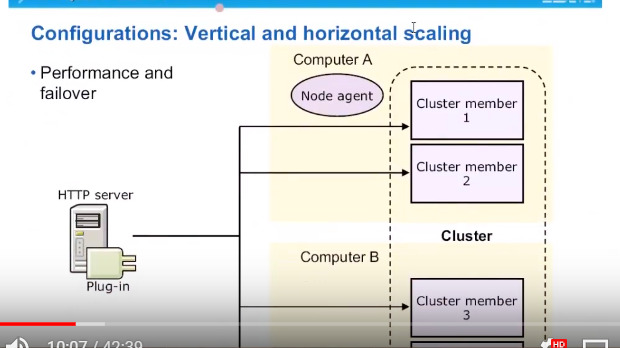


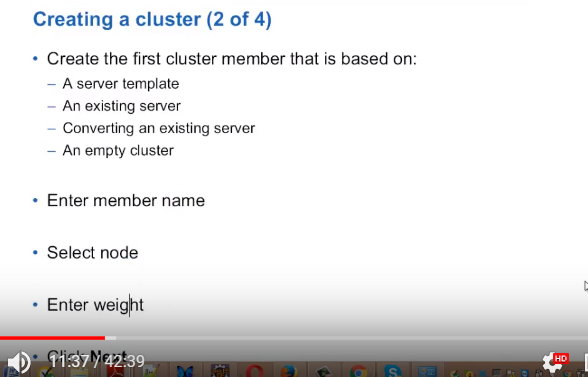


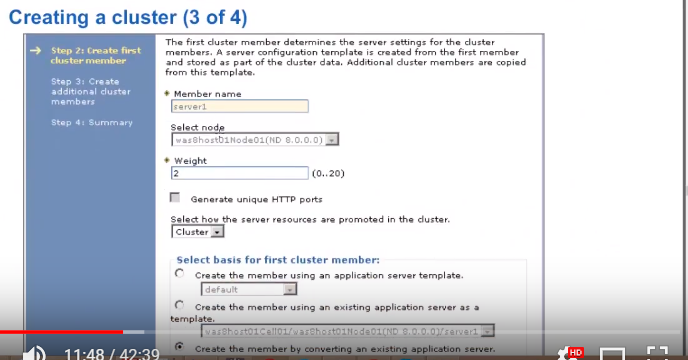


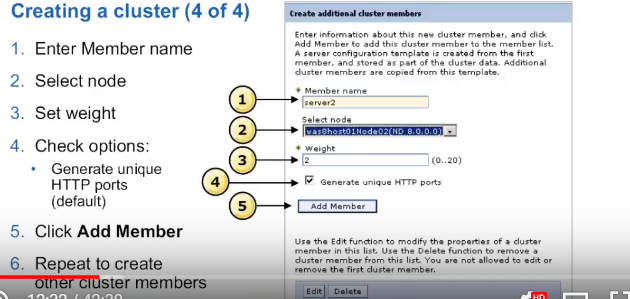


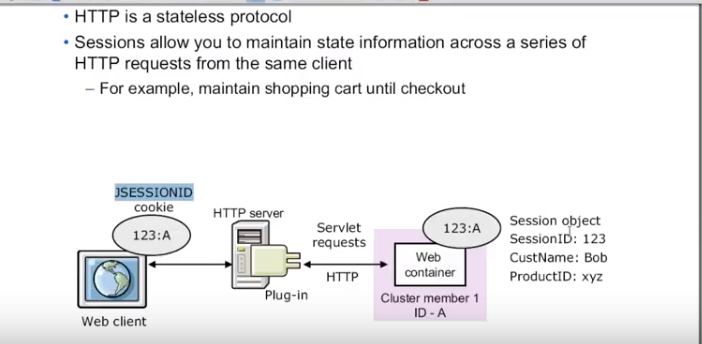


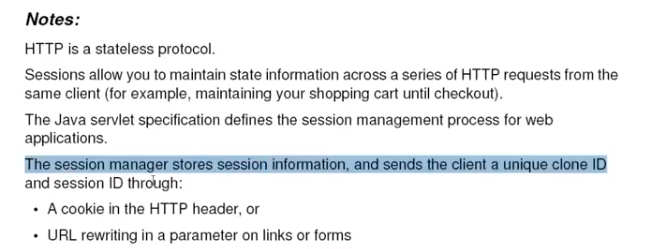




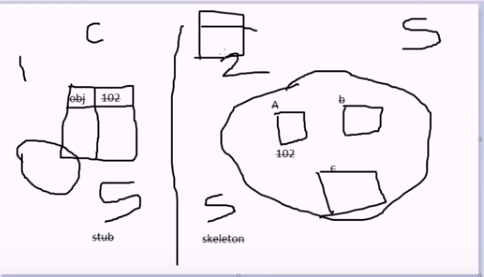


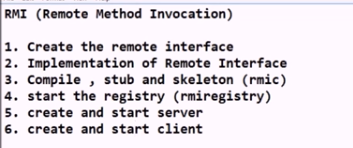






# What is Stub and Skeleton





Examples:

