**Chapter 10:Class Packaging**

In this tutorai we will learn how to import package in class and use of packages.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | |  | | --- | | **0. Import classes and packages**    :    [**Link**](http://www.javatpoint.com/package) |   Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.  A Package can be defined as a grouping of related types(classes, interfaces, enumerations and annotations ) providing access protection and name space management.  Some of the existing packages in Java are::   * **java.lang** - bundles the fundamental classes * **java.io** - classes for input , output functions are bundled in this package   Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, annotations are related.  Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.  Creating a package:  When creating a package, you should choose a name for the package and put a **package**statement with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.  The **package** statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.  If a package statement is not used then the class, interfaces, enumerations, and annotation types will be put into an unnamed package.  Example:  Let us look at an example that creates a package called **animals**. It is common practice to use lowercased names of packages to avoid any conflicts with the names of classes, interfaces.  Put an interface in the package *animals*:  /\* File name : Animal.java \*/  package animals;  interface Animal {  public void eat();  public void travel();  }  Now, put an implementation in the same package *animals*:  package animals;  /\* File name : MammalInt.java \*/  public class MammalInt implements Animal{  public void eat(){  System.out.println("Mammal eats");  }  public void travel(){  System.out.println("Mammal travels");  }  public int noOfLegs(){  return 0;  }  public static void main(String args[]){  MammalInt m = new MammalInt();  m.eat();  m.travel();  }  }  Now, you compile these two files and put them in a sub-directory called **animals** and try to run as follows:  $ mkdir animals  $ cp Animal.class MammalInt.class animals  $ java animals/MammalInt  Mammal eats  Mammal travels  The import Keyword:  If a class wants to use another class in the same package, the package name does not need to be used. Classes in the same package find each other without any special syntax.  Example:  Here, a class named Boss is added to the payroll package that already contains Employee. The Boss can then refer to the Employee class without using the payroll prefix, as demonstrated by the following Boss class.  package payroll;  public class Boss  {  public void payEmployee(Employee e)  {  e.mailCheck();  }  }  What happens if Boss is not in the payroll package? The Boss class must then use one of the following techniques for referring to a class in a different package.   * The fully qualified name of the class can be used. For example:   payroll.Employee   * The package can be imported using the import keyword and the wild card (\*). For example:   import payroll.\*;   * The class itself can be imported using the import keyword. For example:   import payroll.Employee;  **Note:** A class file can contain any number of import statements. The import statements must  appear after the package statement and before the class declaration.  The Directory Structure of Packages:  Two major results occur when a class is placed in a package:   * The name of the package becomes a part of the name of the class, as we just discussed in the previous section. * The name of the package must match the directory structure where the corresponding bytecode resides.   Here is simple way of managing your files in Java:  Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is **.java**. For example:  // File Name : Car.java  package vehicle;  public class Car {  // Class implementation.  }  Now, put the source file in a directory whose name reflects the name of the package to which the class belongs:  ....\vehicle\Car.java  Now, the qualified class name and pathname would be as below:   * Class name -> vehicle.Car * Path name -> vehicle\Car.java (in windows)   In general, a company uses its reversed Internet domain name for its package names. Example: A company's Internet domain name is apple.com, then all its package names would start with com.apple. Each component of the package name corresponds to a subdirectory.  Example: The company had a com.apple.computers package that contained a Dell.java source file, it would be contained in a series of subdirectories like this:  ....\com\apple\computers\Dell.java  At the time of compilation, the compiler creates a different output file for each class, interface and enumeration defined in it. The base name of the output file is the name of the type, and its extension is **.class**  For example:  // File Name: Dell.java  package com.apple.computers;  public class Dell{    }  class Ups{    }  Now, compile this file as follows using -d option:  $javac -d . Dell.java  This would put compiled files as follows:  .\com\apple\computers\Dell.class  .\com\apple\computers\Ups.class  You can import all the classes or interfaces defined in *\com\apple\computers\* as follows:  import com.apple.computers.\*;  Like the .java source files, the compiled .class files should be in a series of directories that reflect the package name. However, the path to the .class files does not have to be the same as the path to the .java source files. You can arrange your source and class directories separately, as:  <path-one>\sources\com\apple\computers\Dell.java  <path-two>\classes\com\apple\computers\Dell.class  By doing this, it is possible to give the classes directory to other programmers without revealing your sources. You also need to manage source and class files in this manner so that the compiler and the Java Virtual Machine (JVM) can find all the types your program uses.  The full path to the classes directory, <path-two>\classes, is called the class path, and is set with the CLASSPATH system variable. Both the compiler and the JVM construct the path to your .class files by adding the package name to the class path.  Say <path-two>\classes is the class path, and the package name is com.apple.computers, then the compiler and JVM will look for .class files in <path-two>\classes\com\apple\compters.  A class path may include several paths. Multiple paths should be separated by a semicolon (Windows) or colon (Unix). By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in the class path.  Set CLASSPATH System Variable:  To display the current CLASSPATH variable, use the following commands in Windows and UNIX (Bourne shell):   * In Windows -> C:\> set CLASSPATH * In UNIX -> % echo $CLASSPATH   To delete the current contents of the CLASSPATH variable, use :   * In Windows -> C:\> set CLASSPATH= * In UNIX -> % unset CLASSPATH; export CLASSPATH   To set the CLASSPATH variable:   * In Windows -> set CLASSPATH=C:\users\jack\java\classes * In UNIX -> % CLASSPATH=/home/jack/java/classes; export CLASSPATH | | |  | | --- | | **0. Usage of packages**    :    [**Link**](http://www.tutorialspoint.com/java/java_packages.htm) | | |  |

# Java Package

1. [Java Package](http://www.javatpoint.com/package)
2. [Example of package](http://www.javatpoint.com/package#packageex)
3. [Accessing package](http://www.javatpoint.com/package#packageaccess)
   1. [By import packagename.\*](http://www.javatpoint.com/package#packageaccess1)
   2. [By import packagename.classname](http://www.javatpoint.com/package#packageaccess2)
   3. [By fully qualified name](http://www.javatpoint.com/package#packageaccess3)
4. [Subpackage](http://www.javatpoint.com/package#packagesub)
5. [Sending class file to another directory](http://www.javatpoint.com/package#packageanotherdirectory)
6. [-classpath switch](http://www.javatpoint.com/package#packageclasspathswitch)
7. [4 ways to load the class file or jar file](http://www.javatpoint.com/package#packagewaystoload)
8. [How to put two public class in a package](http://www.javatpoint.com/package#packagetwopublic)
9. [Static Import](http://www.javatpoint.com/package#packagestaticimport)
10. [Package class](http://www.javatpoint.com/package-class)

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

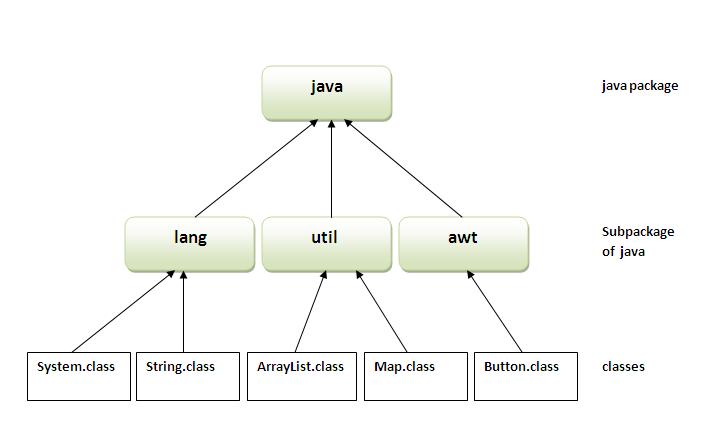
Here, we will have the detailed learning of creating and using user-defined packages.

## Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.



## Simple example of java package

The **package keyword** is used to create a package in java.

1. //save as Simple.java
2. **package** mypack;
3. **public** **class** Simple{
4. **public** **static** **void** main(String args[]){
5. System.out.println("Welcome to package");
6. }
7. }

## How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

## How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

|  |
| --- |
| **To Compile:** javac -d . Simple.java |
| **To Run:** java mypack.Simple |

Output:Welcome to package

|  |
| --- |
| The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder. |

## How to access package from another package?

|  |
| --- |
| There are three ways to access the package from outside the package.   1. import package.\*; 2. import package.classname; 3. fully qualified name. |

#### 1) Using packagename.\*

|  |
| --- |
| If you use package.\* then all the classes and interfaces of this package will be accessible but not subpackages. |

|  |
| --- |
| The import keyword is used to make the classes and interface of another package accessible to the current package. |

## Example of package that import the packagename.\*

1. //save by A.java
3. **package** pack;
4. **public** **class** A{
5. **public** **void** msg(){System.out.println("Hello");}
6. }
7. //save by B.java
9. **package** mypack;
10. **import** pack.\*;
12. **class** B{
13. **public** **static** **void** main(String args[]){
14. A obj = **new** A();
15. obj.msg();
16. }
17. }

Output:Hello

#### 2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

## Example of package by import package.classname

1. //save by A.java
3. **package** pack;
4. **public** **class** A{
5. **public** **void** msg(){System.out.println("Hello");}
6. }
7. //save by B.java
9. **package** mypack;
10. **import** pack.A;
12. **class** B{
13. **public** **static** **void** main(String args[]){
14. A obj = **new** A();
15. obj.msg();
16. }
17. }

Output:Hello

#### 3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

## Example of package by import fully qualified name

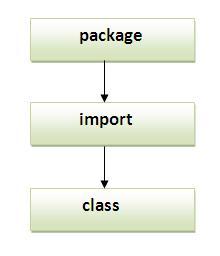
1. //save by A.java
3. **package** pack;
4. **public** **class** A{
5. **public** **void** msg(){System.out.println("Hello");}
6. }
7. //save by B.java
9. **package** mypack;
10. **class** B{
11. **public** **static** **void** main(String args[]){
12. pack.A obj = **new** pack.A();//using fully qualified name
13. obj.msg();
14. }
15. }

Output:Hello

#### Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

#### Note: Sequence of the program must be package then import then class.



## Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

#### The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

### Example of Subpackage

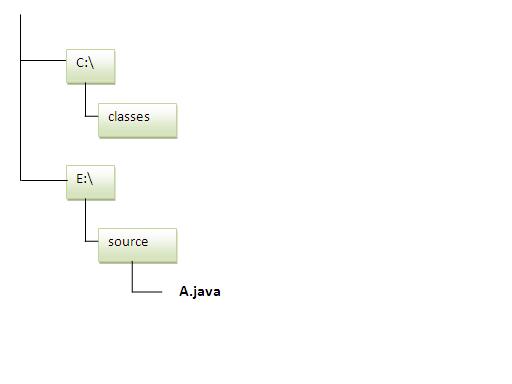
1. **package** com.javatpoint.core;
2. **class** Simple{
3. **public** **static** **void** main(String args[]){
4. System.out.println("Hello subpackage");
5. }
6. }

|  |
| --- |
| **To Compile:** javac -d . Simple.java |
| **To Run:** java com.javatpoint.core.Simple |

Output:Hello subpackage

## How to send the class file to another directory or drive?

|  |
| --- |
| There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example: |



1. //save as Simple.java
3. **package** mypack;
4. **public** **class** Simple{
5. **public** **static** **void** main(String args[]){
6. System.out.println("Welcome to package");
7. }
8. }

### To Compile:

|  |
| --- |
| **e:\sources> javac -d c:\classes Simple.java** |

### To Run:

|  |
| --- |
| To run this program from e:\source directory, you need to set classpath of the directory where the class file resides. |
| **e:\sources> set classpath=c:\classes;.;** |
| **e:\sources> java mypack.Simple** |

### Another way to run this program by -classpath switch of java:

|  |
| --- |
| The -classpath switch can be used with javac and java tool. |

|  |
| --- |
| To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example: |
| **e:\sources> java -classpath c:\classes mypack.Simple** |

Output:Welcome to package

### Ways to load the class files or jar files

|  |
| --- |
| There are two ways to load the class files temporary and permanent. |

* Temporary
  + By setting the classpath in the command prompt
  + By -classpath switch
* Permanent
  + By setting the classpath in the environment variables
  + By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

#### Rule: There can be only one public class in a java source file and it must be saved by the public class name.

1. //save as C.java otherwise Compilte Time Error
3. **class** A{}
4. **class** B{}
5. **public** **class** C{}

### How to put two public classes in a package?

|  |
| --- |
| If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example: |

1. //save as A.java
3. **package** javatpoint;
4. **public** **class** A{}
5. //save as B.java
7. **package** javatpoint;
8. **public** **class** B{}

**Chapter 11:Collections**

In this tutorial we will learn the Java Collection Framework. We will see the API using Collection. We will see the difference between Comparable and Comparator interface. Also learn the concepts of equals() and hashcode() methods.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  | | --- | | **0. Need for Collections.**    :    [**Link**](http://www.jusfortechies.com/java/core-java/collections/collections.php) |   Here is the basic question  *What are Collections?*  A collection - sometimes called a container - is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group, such as a collection of strings, or a mapping of names to addresses.  *Then how are collections (collection implementation classes) different from arrays?*  Nope, they are not different. Array is nothing but a collection and our definition above proves that. Infact the collection implementation classes like ArrayList internally uses arrays to store objects. Vector, Hashtable and array are collection implementations in earlier versions (pre 1.2) of the Java Platform which unfortunately were not easy to extend, and did not implement a standard member interface. Well, they are not part of JDK collections framework but still they are collections.  *Then why do we need Collections Framework?*  All three of these collections viz, Vector, Hashtable, array have different methods and syntax for accessing members: arrays use the square bracket ([]) symbols, Vector uses the elementAt method, and Hashtable uses get and put methods. These differences have forced programmers to implement their own inconsistent collections - some imitate the Vector access methods and some imitate the Enumeration interface. To make it worse, most of the Vector methods are marked as final; that is, you cannot extend the Vector class to implement a similar sort of collection. We could create a collection class that looked like a Vector and acted like a Vector, but it couldn't be passed to a method that takes a Vector as a parameter. Finally, none of the collections (array, Vector or Hashtable) implements a standard member access interface. When programmers develop algorithms (like sorting) to manipulate collections, what object should be passed to the algorithm - is it an array or a Vector or implement both interfaces? A lot of such questions pop up.  Thankfully, the Java Collections Framework (JCF) solves these problems and offers a number of advantages over using no framework or using the Vector and Hashtable  *What is Java Collections Framework (JCF)?*  A collections framework is a unified architecture for representing and manipulating collections. It provides a well-designed set of interfaces and classes for storing and manipulating groups of data as a single unit, a collection. The framework provides a convenient API to many of the abstract data types familiar from computer science data structure curriculum: maps, sets, lists, trees, arrays, hashtables and other collections. Because of their object-oriented design, the Java classes in the Collections Framework encapsulate both the data structures and the algorithms associated with these abstractions. The framework provides a standard programming interface to many of the most common abstractions, without burdening the programmer with too many procedures and interfaces. The operations supported by the collections framework nevertheless permit the programmer to easily define higher level data abstractions, such as stacks, queues, and thread-safe collections.  The main design goal of JCF was to produce API that was reasonably small, both in size, and, more importantly, in "conceptual weight". There are only 14 collection interfaces and the most basic interface is "Collection". To keep the number of core interfaces small, the interfaces do not attempt to capture such subtle distinctions as mutability, modifiability, and resizability. Many of the modification methods in the collection interfaces are labeled optional, allowing implementations to throw an UnsupportedOperationException to indicate that they do not support a specified optional operation. An interface contains a method only if either (a) it is a fundamental operation or (b) there is a compelling perfomance reason why an important implementation would want to override it.  All reasonalble representations of collections should interoperate well. Hence the framework includes methods to allow collections to be dumped into arrays, arrays to be viewed as collections, and maps to be viewed as collections.  *What does Java Collections framework (JCF) contain?*   * Interfaces: These are abstract data types that represent different types of collections, such as sets, lists and maps. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy. * General-purpose Implementations: Primary implementations of the collection interfaces. * Legacy Implementations: The collection classes from earlier releases, Vector and Hashtable, have been retrofitted to implement the collection interfaces. * Special-purpose Implementations: Implementations designed for use in special situations. These implementations display nonstandard performance characteristics, usage restrictions, or behavior. * Concurrent Implementations: Implementations designed for highly concurrent use. * Wrapper Implementations: Add functionality, such as synchronizations, to other implementations. * Convenience Implemenations: High-performance "mini-implementations" of the collection interfaces. * Abstract Implementations: Partial implementations of the collection interfaces to facilitate custom implementations. * Algorithms: Static methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality. * Infrastructure: Interfaces that provide essential suport for the collection interfaces. * Array Utilities: Utility functions for arrays of primitives and reference objects. Not, strictly speaking, a part of the Collections Framework, this functionality was added to the Java platform at the same time and relies on some of the same infrastructure.   *So what is the benefit of going for Java Collections Framework (JCF) instead of self defined and implemented classes or array?*  Well, Java Collections Framework   * Reduces programming effort: By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than writing them all by yourself. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs. * Increases program speed and quality (performance): Java Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. As the various implementations of each interface are interchangeable, programs can be easily tuned by switching collection implementations. Because you're freed from writing hell lot of your own data structures, you'll have more time to devote to improving programs' quality and performance. * Provides interoperability between unrelated APIs: The collection interfaces provides interoperability between unrelated APIs by establishing common language to pass collections back and forth. * Reduces effort to learn new APIs: Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away. * Reduces effort to design and implement new APIs: This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces. * Fosters software reuse: New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.   The following diagrams shows the collections framework interface hierarchy.   There are fourteen collection interfaces. Set, List, SortedSet, NavigableSet, Queue, Deque, BlockingQueue and BlockingDeque interfaces extend Collection. Map, SortedMap, NavigableMap, ConcurrentMap and ConcurrentNavigableMap interfaces do not extend Collection interface, as they represent mappings rather than true collections. However, these interfaces contain collection-view operations, which allow them to be manipulated as collections. Some collection implementations may restrict what elements may be stored (Ex: non-null values, or specific type elements, etc) and attempting to add or remove or test for the presence of an element that violates an implementation's restrictions results in a runtime exception, typically a ClassCastException, an IllegalArgumentException or a NullPointerException.    The following table shows the general-purpose collection implementations hierarchy.   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  | | **Implementations** | | | | | | **Hash Table** | **Resizable Array** | **Balanced Tree** | **Linked List** | **Hash Table & Linked List** | | **Interfaces** | **Set** | HashSet |  | TreeSet |  | LinkedHashSet | | **List** |  | ArrayList |  | LinkedList |  | | **Deque** |  | ArrayDeque |  | LinkedList |  | | **Map** | HashMap |  | TreeMap |  | LinkedHashMap |     Classes that implement the collection interfaces typically have names of the form <implementation-style><Interface>. The general-purpose implementations are unsynchronized, but the *Collections*class contains static factories called synchronization wrappers that may be used to add synchronization to many unsynchronized collections. The AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList and AbstractMap classes provide skeletal implentations of the core collection interfaces, to minimize the effort required to implement them. You can extent these classes and create you own collection implementation class.  *Few interesting points to note:*   * Collection interfaces do not extend Serializable or Cloneable interface   Many Collection implementations will have a public clone method, but it would be mistake to require it of all collections. For example, it wont make sense to clone a Collection that is backed by a terabyte SQL database. Similar arguments hold for serializable. If the client does not know the actual type of a Collection, it is much more flexible and less error prone to have the client decide what type of Collection is desired, create an empty Collection of this type, and use the addAll method to copy the elements of the original collection into the new one.   * Collection classes do not support immutalibility directly (supports through Collection class)   The main reason is to keep the Collections framework simple. So for a second assume that we want to add notion of modifiability to Collection hierarchy. First - you need four new interfaces: ModifiableCollection, ModifiableList, ModifiableSet, ModifiableMap. Then a new Iterator interface for use with unmodifiable Collections, that does not contain the remove operation. Third - you need to add variable-size lists: VariableSizeList and VariableSizeMap. This will go on and on and you will end up with whole bunch of interfaces. And what about immutable Collections, as opposed to unmodifiable ones? i.e., Collections that cannot be changed by the client and will never change for any other reason. Adding this support to the type hierarchy required four more interfaces. Considering all these, it was a sound engineering compromise to avoid the issue by providing a very small set of core interfaces that can throw a runtime exception. | | |  | | --- | | **0. The Collection Framework**    :    [**Link**](http://www.tutorialspoint.com/java/java_collections.htm)  [**Link**](http://www.comscigate.com/courses/UBC211/assignments/a1/a1.html) |  Java Collections Framework Advertisements  [Previous Page](http://www.tutorialspoint.com/java/java_data_structures.htm)  [Next Page](http://www.tutorialspoint.com/java/java_generics.htm)  Prior to Java 2, Java provided ad hoc classes such as **Dictionary, Vector, Stack**, and**Properties** to store and manipulate groups of objects. Although these classes were quite useful, they lacked a central, unifying theme. Thus, the way that you used Vector was different from the way that you used Properties.  The collections framework was designed to meet several goals.   * The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hashtables) are highly efficient. * The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability. * Extending and/or adapting a collection had to be easy.   Towards this end, the entire collections framework is designed around a set of standard interfaces. Several standard implementations such as **LinkedList, HashSet,** and **TreeSet**, of these interfaces are provided that you may use as-is and you may also implement your own collection, if you choose.  A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:   * **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy. * **Implementations, i.e., Classes:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures. * **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.   In addition to collections, the framework defines several map interfaces and classes. Maps store key/value pairs. Although maps are not *collections* in the proper use of the term, but they are fully integrated with collections. The Collection Interfaces: The collections framework defines several interfaces. This section provides an overview of each interface:   |  |  | | --- | --- | | **SN** | **Interfaces with Description** | | 1 | [**The Collection Interface**](http://www.tutorialspoint.com/java/java_collection_interface.htm)  This enables you to work with groups of objects; it is at the top of the collections hierarchy. | | 2 | [**The List Interface**](http://www.tutorialspoint.com/java/java_list_interface.htm)  This extends **Collection** and an instance of List stores an ordered collection of elements. | | 3 | [**The Set**](http://www.tutorialspoint.com/java/java_set_interface.htm)  This extends Collection to handle sets, which must contain unique elements | | 4 | [**The SortedSet**](http://www.tutorialspoint.com/java/java_sortedset_interface.htm)  This extends Set to handle sorted sets | | 5 | [**The Map**](http://www.tutorialspoint.com/java/java_map_interface.htm)  This maps unique keys to values. | | 6 | [**The Map.Entry**](http://www.tutorialspoint.com/java/java_mapentry_interface.htm)  This describes an element (a key/value pair) in a map. This is an inner class of Map. | | 7 | [**The SortedMap**](http://www.tutorialspoint.com/java/java_sortedmap_interface.htm)  This extends Map so that the keys are maintained in ascending order. | | 8 | [**The Enumeration**](http://www.tutorialspoint.com/java/java_enumeration_interface.htm)  This is legacy interface and defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superceded by Iterator. |  The Collection Classes: Java provides a set of standard collection classes that implement Collection interfaces. Some of the classes provide full implementations that can be used as-is and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.  The standard collection classes are summarized in the following table:   |  |  | | --- | --- | | **SN** | **Classes with Description** | | 1 | **AbstractCollection**  Implements most of the Collection interface. | | 2 | **AbstractList**  Extends AbstractCollection and implements most of the List interface. | | 3 | **AbstractSequentialList**  Extends AbstractList for use by a collection that uses sequential rather than random access of its elements. | | 4 | [**LinkedList**](http://www.tutorialspoint.com/java/java_linkedlist_class.htm)  Implements a linked list by extending AbstractSequentialList. | | 5 | [**ArrayList**](http://www.tutorialspoint.com/java/java_arraylist_class.htm)  Implements a dynamic array by extending AbstractList. | | 6 | **AbstractSet**  Extends AbstractCollection and implements most of the Set interface. | | 7 | [**HashSet**](http://www.tutorialspoint.com/java/java_hashset_class.htm)  Extends AbstractSet for use with a hash table. | | 8 | [**LinkedHashSet**](http://www.tutorialspoint.com/java/java_linkedhashset_class.htm)  Extends HashSet to allow insertion-order iterations. | | 9 | [**TreeSet**](http://www.tutorialspoint.com/java/java_treeset_class.htm)  Implements a set stored in a tree. Extends AbstractSet. | | 10 | **AbstractMap**  Implements most of the Map interface. | | 11 | [**HashMap**](http://www.tutorialspoint.com/java/java_hashmap_class.htm)  Extends AbstractMap to use a hash table. | | 12 | [**TreeMap**](http://www.tutorialspoint.com/java/java_treemap_class.htm)  Extends AbstractMap to use a tree. | | 13 | [**WeakHashMap**](http://www.tutorialspoint.com/java/java_weakhashmap_class.htm)  Extends AbstractMap to use a hash table with weak keys. | | 14 | [**LinkedHashMap**](http://www.tutorialspoint.com/java/java_linkedhashmap_class.htm)  Extends HashMap to allow insertion-order iterations. | | 15 | [**IdentityHashMap**](http://www.tutorialspoint.com/java/java_identityhashmap_class.htm)  Extends AbstractMap and uses reference equality when comparing documents. |   The *AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList* and *AbstractMap*classes provide skeletal implementations of the core collection interfaces, to minimize the effort required to implement them.  The following legacy classes defined by java.util have been discussed in previous tutorial:   |  |  | | --- | --- | | **SN** | **Classes with Description** | | 1 | [**Vector**](http://www.tutorialspoint.com/java/java_vector_class.htm)  This implements a dynamic array. It is similar to ArrayList, but with some differences. | | 2 | [**Stack**](http://www.tutorialspoint.com/java/java_stack_class.htm)  Stack is a subclass of Vector that implements a standard last-in, first-out stack. | | 3 | [**Dictionary**](http://www.tutorialspoint.com/java/java_dictionary_class.htm)  Dictionary is an abstract class that represents a key/value storage repository and operates much like Map. | | 4 | [**Hashtable**](http://www.tutorialspoint.com/java/java_hashtable_class.htm)  Hashtable was part of the original java.util and is a concrete implementation of a Dictionary. | | 5 | [**Properties**](http://www.tutorialspoint.com/java/java_properties_class.htm)  Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String. | | 6 | [**BitSet**](http://www.tutorialspoint.com/java/java_bitset_class.htm)  A BitSet class creates a special type of array that holds bit values. This array can increase in size as needed. |  The Collection Algorithms: The collections framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the Collections class.  Several of the methods can throw a **ClassCastException**, which occurs when an attempt is made to compare incompatible types, or an **UnsupportedOperationException**, which occurs when an attempt is made to modify an unmodifiable collection.  Collections define three static variables: EMPTY\_SET, EMPTY\_LIST, and EMPTY\_MAP. All are immutable.   |  |  | | --- | --- | | **SN** | **Algorithms with Description** | | 1 | [**The Collection Algorithms**](http://www.tutorialspoint.com/java/java_collection_algorithms.htm)  Here is a list of all the algorithm implementation. |  How to use an Iterator ? Often, you will want to cycle through the elements in a collection. For example, you might want to display each element.  The easiest way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface.  Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list and the modification of elements.   |  |  | | --- | --- | | **SN** | **Iterator Methods with Description** | | 1 | [**Using Java Iterator**](http://www.tutorialspoint.com/java/java_using_iterator.htm)  Here is a list of all the methods with examples provided by Iterator and ListIterator interfaces. |  How to use a Comparator ?  |  |  | | --- | --- | | **SN** | **Iterator Methods with Description** | | 1 | [**Using Java Comparator**](http://www.tutorialspoint.com/java/java_using_comparator.htm)  Here is a list of all the methods with examples provided by Comparator Interface. |   Both TreeSet and TreeMap store elements in sorted order. However, it is the comparator that defines precisely what *sorted order* means.  This interface lets us sort a given collection any number of different ways. Also this interface can be used to sort any instances of any class (even classes we cannot modify). Summary: The Java collections framework gives the programmer access to prepackaged data structures as well as to algorithms for manipulating them.  A collection is an object that can hold references to other objects. The collection interfaces declare the operations that can be performed on each type of collection.  The classes and interfaces of the collections framework are in package java.util. Problem specification You will produce a music library with functionality similar to programs such as Microsoft Windows Media Player and Apple iTunes. Your program will allow the user to:   * View a list of all the songs in the library and see information about each song, including the album on which it appears and the artist(s) who performed it. * View a list of all the artists in the library. * View a list of all the albums in the library. * Edit any information in the library. This includes changing the title of songs and albums and changing the names of artists. * View a list of all the songs by a particular artist. * View a list of all the songs on a particular album. * Add songs to and remove them from the library. * Add artists to and remove them from songs. * Add songs to and remove them from albums. * Play songs using the provided player.  Organization Your project will contain 4 packages:   * player: this package (and it's sub-packages) are provided for you and you will use them to play songs * util: this package contains a class you can use to load descriptive information stored in MP3 files * ui: this package contains a test driver you can use to test your library * library: this is the package you will implement   For this assignment, you will not have to use the player or util packages directly as they will be used by the test driver we provide. Classes You Will Implement You will implement the 4 classes in the library package as they are specified in the [javadoc](http://www.comscigate.com/courses/UBC211/assignments/resources/javadoc/library/package-summary.html). The following class diagram gives an overview of the package:  http://www.comscigate.com/courses/UBC211/assignments/a1/images/library_overview.png   * Each instance of the Song class will correspond to a unique file on disk and will store the following pieces of information:   + The path to the file on disk   + The title of the song   + The duration of the song   + The track number * Each instance of the Artist class will store the name of a person (eg. "Lucinda Williams") or group (eg. "Ben Harper & the Innocent Criminals", "Radiohead") that performs Songs. There will be no distinction between groups and individual people. * Each instance of the Album class will represent an album - a collection of Songs ordered by track number (where each song on an album has a different track number) - and will store the album title. In addition, there will be at least one album in the system at all times (a "default" album) and every song will be associated with exactly one album as shown above. Any song for which the user has not entered an album will belong to the default album. * There will be a single instance of the Library class in the system which will maintain collections of the other three classes and provide methods to create and destroy them that ensure consistency between the library and the songs, artists, and albums. * Each class will also implement the associations depicted above.   You must implement all the methods described in the javadoc and will probably need to add some private methods of your own. Note that you are **not** allowed to modify the classes in the uipackage. The code we provided **must** work with your implementation of the library package. Library as a Singleton As stated in the Javadoc, you must implement the Library class as a singleton. A singleton class has the following features:   * Only a single instance of the class may be created and this instance can be accessed from anywhere. This is appropriate for the Library class because it would make no sense (in this context) to have multiple libraries, and because many classes will want to manipulate the library. * To ensure that there is exactly one instance of the class, the constructor is **private** and is called once from within the class to initialize the single instance, which will be kept as a static field of the class. * To allow access to the single instance, the class defines a static method called getInstance.   Here is an example:  public class Singleton  {  // this is the only time the constructor is ever called  private static Singleton instance = new Singleton();  // This is the only constructor and it is private  private Singleton()  {  // standard constructor code goes here  }  // Return the single instance of this class. Notice that  // every call to this method returns the same instance; no new  // instances are created.  public static Singleton getInstance()  {  return instance;  }  }  To access the singleton class, other classes would call Singleton.getInstance(). Collections The "many" ends of the above associations must be implemented using members of the Java Collections Framework. You will be graded (in part) on which collections you choose to use. For each case, you should ask yourself the following questions:   * Which is more appropriate, a Collection or a Map? * If I use a Collection, should I use one that permits duplicates? * Should I use a collection that does or does not keep its elements in a specific order? * Which implementation will be most efficient, based on the expected usage of the collection?   Here you should also consider whether the features of a particular collection are worth the cost. For example, you may want to use a Set to maintain a collection that is not allowed to contain duplicates, however, the restriction on modifying members of a Set might be inconvenient and you may not expect this collection to contain more than a few elements. In this case it may be much more efficient to use a List and manually ensure that you never insert duplicate elements. A Note On Terminology In this document, the word "collection" written in plain text refers to a member of the Java Collections Framework (ie. a class that implements either java.util.Collection or java.util.Map). The word "Collection," capitalized and written like this refers to java.util.Collection or one of its implementations. In other words, a Collection is a collection that is not a Map. This is the same convention used in the Java API and the Java Tutorial. Other Specific Requirements In addition to everything stated above (and below) this section you should keep the following in mind:   * When editing information in the library or adding songs, users of your program will specify Artists and Albums by entering their names/titles. * We assume the path to songs will not change, that is, if the user renamed or moved a file they would need to remove the corresponding Song from the Library and create a new Song. * The library will not be allowed to contain two Albums with the same title, two Artists with the same name, or two Songs representing the same file. * Any method that returns a collection of Albums should return them in alphabetical order by title. * Any method that returns a collection of Artists should return them in alphabetical order by name, with one exception: invoking getArtists() on a Song should return them in the order they were added to the Song. * The order in which Songs are returned is irrelevant (since it is not clear by which attribute the user is most likely to want their songs ordered - we can leave this decision up to the user interface) with the exception that the getSongs() method of the Album class must return its Songs ordered by track number.  Testing Your Library You should use the TestDriver class in the ui package to test your library. It takes a single, optional command line argument: a directory containing MP3 files. Multiple directories can be specified by separating them with a colon (:) on UNIX systems or a semicolon (;) on Windows systems. When working in the Linux labs you can use /home/c/cs211/World/assignments/mp3, which contains a number of MP3 files you can use for testing purposes.  TestDriver will then create a Library containing Songs, Artists and Albums based on the information it reads from the [ID3 tags](http://www.id3.org/) of the files in the specified directory. You can use TestDriver to view the contents of the library, to play, add, and remove songs and to edit the information contained in the library. The interface provided is menu-based. In assignment 3, you will replace it with a much nicer-looking graphical user interface.  When you exit the test driver, it will save the library (including any changes you make) to a file called MusicLibrary.xml in your MusicLibrary directory. When the test driver is started again, it will first load the contents of the MusicLibrary.xml file and then add any other songs in the directories specified on the command line. To revert to the information stored in the MP3 files, simply delete the MusicLibrary.xml file and restart the test driver.  You should not assume that your classes are bug-free just because the test driver works correctly, but it will probably catch any major bugs. Still, you should test each method and each class as you write it; you do not want to find yourself in the position of having spent many hours writing a complete system which completely fails to work. Where to start  1. You should first read and understand the javadoc for the library package in its entirety because it specifies in detail what you need to do for this assignment. Think about the interaction between the methods of different classes, remembering that a method that creates or destroys a bi-directional association between classes must ensure that the two ends of the association are consistent with each other. Note that you can add methods with default (package level) visibility which can only be called by classes within the library package - this restriction on how a method will be called relaxes the requirement that it maintain consistency provided the calling method takes on that obligation. 2. Take some time to review the Java Collections Framework and think about which class would be most appropriate for each of the associations. 3. Implement the system. A good place to start is the Album class as it is probably the simplest and least coupled class in the package.  Deliverables  1. The library package you created, including the following:    * Library.java    * Artist.java    * Song.java    * Album.java   You should not have written or modified any code outside of these 4 files.   1. A text file, named README, that includes:    * Your name, student id#, and lab section    * Any assumptions and rationales you made.    * A list of known bugs and issues.    * Any other points you want the TA/Marker to know.   This file **must** be human-readable. Microsoft Word documents, Excel spreadsheets and other encrypted documents are **not** acceptable. Handin Instructions  1. Create the directory ~/cs211/a1.%, replacing the % in the directory name by the appropriate letter for your lab section (a, b, c, etc). 2. Copy all of the files listed in the Deliverables section, including your README file, to your a1.% directory. 3. Once you are satisfied that your a1 directory contains all the deliverables and nothing else, run the command handin cs211 a1.% to submit your assignment. You can obtain more information about the handin command by looking [here](http://www.comscigate.com/courses/UBC211/course-info/handin.html). | | |  | | --- | | **0. The Collections API :- The java.util package**    :    [**Link**](http://tutorials.jenkov.com/java-collections/index.html) |  Java Collections Tutorial http://tutorials.jenkov.com/images/layout/small-portrait_mini.jpg  By [Jakob Jenkov](http://jakob.jenkov.com/)   Connect with me:  Rate article:  Share article:  The Java Collections API's provide Java developers with a set of classes and interfaces that makes it easier to handle collections of objects. In a sense Collection's works a bit like arrays, except their size can change dynamically, and they have more advanced behaviour than arrays.  Rather than having to write your own collection classes, Java provides these ready-to-use collection classes for you. This tutorial will look closer at the Java Collection's, as they are also sometimes referred to, and more specifically the Java Collections available in Java 6.  The purpose of this tutorial is to give you an overview of the Java Collection classes. Thus it will not describe each and every little detail of the Java Collection classes. But, once you have an overview of what is there, it is much easier to read the rest in the JavaDoc's afterwards.  Most of the Java collections are located in the java.util package. Java also has a set of concurrent collections in the java.util.concurrent package. This tutorial will not describe the concurrent collections. These will be described in their own tutorial some time in the future. Overview of Java Collections To help you get an overview of the Java Collections classes and interfaces, the first text in this Java Collections tutorial is the [**Overview of Interfaces**](http://tutorials.jenkov.com/java-collections/overview.html) text. The Central Java Collection Interfaces The third and fourth text explains the two central interfaces: [**java.util.Collection**](http://tutorials.jenkov.com/java-collections/collection.html) and [**java.util.Iterable**](http://tutorials.jenkov.com/java-collections/iterable.html). Additionally, the [**java.util.Map**](http://tutorials.jenkov.com/java-collections/map.html) is central too. Java Collections and Generics The fifth text in this Java Collections tutorial covers how to use Generics in Java Collections. Generics is very useful when working with Java's Collection classes. Java Collections and the equals() and hashCode() Methods The last two texts in this Java Collections tutorial explains the central role the two java.lang.Objectmethods equals() and hashCode() play when using Java Collections. You should read this, if you plan to use Java Collections with your own classes, and thus need to implement equals() and hashCode(). | | |  | | --- | | **0. Collections utility class**    :    [**Link**](http://www.w3resource.com/java-tutorial/java-collections-utility-class.php)  [**Link**](http://www.leepoint.net/notes-java/data/collections/exercises/q-collections-2.html) |  Java Collections Utility Class [**<<Previous**](http://www.w3resource.com/java-tutorial/java-maps.php)[**Next>>**](http://www.w3resource.com/java-tutorial/java-defining-instantiating-and-starting-threads.php) Introduction The Collections utility class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection,  Some useful method in Collections class:   |  |  | | --- | --- | | Method Signature | Description | | Collections.sort(List myList) | Sort the myList (implementation of any List interface) provided in argument in natural ordering. | | Collections.sort(List, comparator c) | Sort the myList(implementation of any List interface) as per comparator c ordering (c class should implement comparator interface) | | Collections.shuffle(List myList) | Puts the elements of myList ((implementation of any List interface)in random order | | Collections.reverse(List myList) | Reverses the elements of myList ((implementation of any List interface) | | Collections.binarySearch(List mlist, T key) | Searches the mlist (implementation of any List interface) for the specified object using the binary search algorithm. | | Collections.copy(List dest, List src) | Copy the source List into dest List. | | Collections.frequency(Collection c, Object o) | Returns the number of elements in the specified collection class c (which implements Collection interface can be List, Set or Queue) equal to the specified object | | Collections.synchronizedCollection(Collection c) | Returns a synchronized (thread-safe) collection backed by the specified collection. |   Lets take example of List sorting using Collection class. We can sort any Collection using “Collections” utility class. i.e.; ArrayList of Strings can be sorted alphabetically using this utility class. ArrayList class itself is not providing any methods to sort. We use Collections class static methods to do this. Below program shows use of reverse(), shuffle(), frequency() methods as well. Java Code [view plaincopy to clipboardprint?](http://www.w3resource.com/java-tutorial/java-collections-utility-class.php)   1. **package** utility; 3. **import** java.util.Collections; 4. **import** java.util.ArrayList; 5. **import** java.util.List; 7. **public** **class** CollectionsDemo { 9. **public** **static** **void** main(String[] args) { 10. List<String>student<String>List = **new** ArrayList(); 11. studentList.add("Neeraj"); 12. studentList.add("Mahesh"); 13. studentList.add("Armaan"); 14. studentList.add("Preeti"); 15. studentList.add("Sanjay"); 16. studentList.add("Neeraj"); 17. studentList.add("Zahir"); 19. System.out.println("Original List " + studentList); 21. Collections.sort(studentList); 22. System.out.println("Sorted alphabetically List " + studentList); 24. Collections.reverse(studentList); 25. System.out.println("Reverse List " + studentList); 26. Collections.shuffle(studentList); 27. System.out.println("Shuffled List " + studentList); 28. System.out.println("Checking occurance of Neeraj: " 29. + Collections.frequency(studentList, "Neeraj")); 30. } 31. }  Output java collection uitlity image1  Using Collections class we can copy one type of collection to other type. Collections provide us copy method to copy all the elements from source to destination. Below program demonstrate the use of copy function. Here size of source collection and destination collection should be same else we will get following exception.  java collection uitlity image2 Java Code [view plaincopy to clipboardprint?](http://www.w3resource.com/java-tutorial/java-collections-utility-class.php)   1. **package** utility; 3. **import** java.util.Collections; 4. **import** java.util.\*; 6. **public** **class** CopyListDemo { 8. **public** **static** **void** main(String[] args) { 9. List <Integer>myFirstList = **new** ArrayList<Integer>(); 10. List <Integer> mySecondList = **new** ArrayList<Integer>(); 11. myFirstList.add(10); 12. myFirstList.add(20); 13. myFirstList.add(20); 14. myFirstList.add(50); 15. myFirstList.add(70);  18. mySecondList.add(11); 19. mySecondList.add(120); 20. mySecondList.add(120); 21. mySecondList.add(150); 22. mySecondList.add(170); 24. System.out.println("First List-"+ myFirstList); 25. System.out.println("Second List-"+ mySecondList); 26. Collections.copy(mySecondList, myFirstList ); 27. System.out.println("Second List After Copy-"+ mySecondList); 28. } 29. }  Output java collection uitlity image3 | | |  | | --- | | **0. Comparable and Comparator**    :    [**Link**](http://javarevisited.blogspot.in/2011/06/comparator-and-comparable-in-java.html)  [**Link**](http://www.roseindia.net/java/java-tips/data/collections/exercises/q-collections-4.shtml) |  [How to use Comparator and Comparable in Java? With example](http://javarevisited.blogspot.in/2011/06/comparator-and-comparable-in-java.html) **Comparator and Comparable in Java Examples**  Difference between Comparator and Comparable in Java is very [popular Java interview question](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html) mostly asked in telephonic round and writing code to sort object using Comparable or Comparator is popular on  written test round of interview.The question was this “How you will sort Employee object based on his EmployeeID and his name” and this involves the use of both Comparable as well as Comparator interface in Java. This post is my revision on Java fundamentals similar to I did about [equals method in Java](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) and  some tips to [override hashCode in Java](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html). All of these methods are fundamentals in Java programming language and correct understanding is must for any Java developer.**Comparators and comparable** in Java are two interfaces which is used to implement sorting in Java. It’s often required to sort objects stored in any collection classes like ArrayList, HashSet or in Array and that time we need to use either  compare() or  compareTo() method defined in java.util.Comparator and java.lang.Comparable. In this Java tutorial we will see example of  Comparator and Comparable to sort object in Java and discuss some best practices around when to use Comparator interface etc. Any way before moving ahead Let’s see some important differences between Comparable and Comparator in Java.   Comparator vs Comparable in Java [Difference between Comparator vs Comparable in Java](http://2.bp.blogspot.com/-wrzDeQGAe1I/TWu8pLuLr4I/AAAAAAAAADE/V017G-6Q61w/s1600/java_logo_50_50.jpg)Here are some of the common differences, which is worth remembering to answer this question if asked during a telephonic or face to face interview:  1) Comparator in Java is defined in java.util package while Comparable interface in Java is defined in java.lang package, which very much says that Comparator should be used as an utility to sort objects which Comparable should be provided by default.  2) Comparator interface in Java has method public int compare (Object o1, Object o2) which returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second. While Comparable interface has method public int compareTo(Object o) which returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.  3) If you see then logical difference between these two is *Comparator in Java* compare two objects provided to him, while Comparable interface compares "this" reference with the object specified. I have shared lot of tips on [how to override compareTo() method](http://javarevisited.blogspot.sg/2011/11/how-to-override-compareto-method-in.html) and avoid some common mistakes programmer makes while implementing Comparable interface.  4) Comparable in Java is used to implement **natural ordering of object**. In Java API String, Date and wrapper classes implements Comparable interface.Its always good practice to override compareTo() for value objects.  5) If any class implement Comparable interface in Java then collection of that object either [List](http://javarevisited.blogspot.sg/2012/04/difference-between-list-and-set-in-java.html) or Array can be sorted automatically by using  Collections.sort() or Arrays.sort() method and object will be sorted based on there natural order defined by CompareTo method.  6)Objects which implement *Comparable in Java*  can be used as keys in a SortedMap like [TreeMap](http://javarevisited.blogspot.sg/2011/12/treemap-java-tutorial-example-program.html) or elements in a SortedSet  for example TreeSet, without specifying any Comparator.  These were combination of some theoretical and practical differences between Comparator and Comparator interface in Java. It does help you to decide when to use Comparator vs Comparable but things will be more clear when we some best practices around using both of these interfaces. Now let’s see an example of Comparator in Java:   Example of using Comparator and Comparable in Java So in Summary if you want to **sort objects based on natural order** then use Comparable in Java and if you want to sort on some other attribute of object then use Comparator in Java. Now to understand these concepts lets see an example or real life coding:  1) There is class called Person, sort the Person based on person\_id, which is primary key in database  2) Sort the Person based on there name.  For a Person class, sorting based on person\_id can be treated as *natural order sorting* and sorting based on name field can be implemented using Comparator interface. To sort based on person\_id we need to implement compareTo() method.  **public** **class** Person **implements** **Comparable** {     **private** **int** person\_id;     **private** **String** name;          /\*\*      \* Compare current person with specified person      \* return zero if person\_id for both person is same       \* return negative if current person\_id is less than specified one      \* return positive if specified person\_id is greater than specified one      \*/   @Override      **public** **int** compareTo(Object o) {         Person p = (Person) o;          **return** **this**.person\_id - o.person\_id ;     }     …. }  Generally you should not use difference of integers to decide output of compareTo method as result of **integer subtraction can overflow**but if you are sure that both operands are positive then its one of the quickest way to compare two objects. See my post [things to remember while overriding compareTo in Java](http://javarevisited.blogspot.sg/2011/11/how-to-override-compareto-method-in.html) for more tips on compareTo.  And for sorting based on person name we can implement compare(Object o1, Object o2) method of Java Comparator class.  /\*\*  \* Comparator implementation which sorts Person objects on person\_id field  \*/ **public** **class** SortByPerson\_ID **implements** **Comparator**{      **public** **int** compare(Object o1, Object o2) {         Person p1 = (Person) o;         Person p2 = (Person) o;          **return** p1.getPersonId() - p2.getPersonId();     } }  Similar guidelines applies while implementing compare() method as well and instead of using subtraction operator, its better to use logical operator to compare whether two integers are equal to, less than or greater than. You can write several types of Java Comparator based upon your need for example  reverseComparator , ANDComparator , ORComparator etc which will return negative or positive number based upon logical results. [String in Java](http://javarevisited.blogspot.sg/2011/07/string-vs-stringbuffer-vs-stringbuilder.html) even provides an special comparator called CASE\_INSENSITIVE\_ORDER, to perform case insensitive comparison of String objects.  **How to Compare String in Java**  [String is immutable in Java](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html) and one of the most used value class. For comparing String in Java we should not be worrying because String implements Comparable interface and provides a lexicographic implementation for CompareTo method which compare two strings based on contents of characters or you can say in lexical order. You just need to call String.compareTo(AnotherString) and Java will determine whether specified String is greater than , equal to or less than current object. See my post [4 example to compare String in Java](http://javarevisited.blogspot.sg/2012/03/how-to-compare-two-string-in-java.html) for alternatives ways of comparing String.  **How to Compare Dates in Java**  Dates are represented by java.util.Date class in Java and like String,  Date also implements Comparable in Java so they will be automatically sorted based on there natural ordering if they got stored in any sorted collection like TreeSet or TreeMap. If you explicitly wants to compare two dates in Java you can call Date.compareTo(AnotherDate) method in Java and it will tell whether specified date is greater than , equal to or less than current String. See my post [3 ways to compare Dates in Java](http://javarevisited.blogspot.sg/2012/02/3-example-to-compare-two-dates-in-java.html) for more alternatives of comparing two dates.  **When to use Comparator and Comparable in Java**  At last let’s see some best practices and recommendation on when to use Comparator or Comparable in Java:  1) If there is a natural or default way of sorting Object already exist during development of Class than use Comparable. This is intuitive and you given the class name people should be able to guess it correctly like Strings are sorted chronically, Employee can be sorted by there Id etc. On the other hand if an Object can be sorted on multiple ways and client is specifying on which parameter sorting should take place than use Comparator interface. for example Employee can again be sorted on name, salary or department and clients needs an API to do that. Comparator implementation can sort out this problem.  2) Some time you write code to sort object of a class for which you are not the original author, or you don't have access to code. In these cases you can not implement Comparable and Comparator is only way to sort those objects.  3) Beware with the fact that How those object will behave if stored in SorteSet or SortedMap like TreeSet and [TreeMap](http://javarevisited.blogspot.sg/2011/12/treemap-java-tutorial-example-program.html). If an object doesn't implement Comparable than while putting them into SortedMap, always provided corresponding Comparator which can provide sorting logic.  4) Order of comparison is very important while implementing Comparable or Comparator interface. for example if you are sorting object based upon name than you can compare first name or last name on any order, so decide it judiciously. I have shared more detailed tips on compareTo on my post how to implement CompareTo in Java.  5) Comparator has a distinct advantage of being self descriptive  for example if you are writing Comparator to compare two Employees based upon there salary than name that comparator as SalaryComparator, on the other hand compareTo()  Read more: <http://javarevisited.blogspot.com/2011/06/comparator-and-comparable-in-java.html#ixzz3gPdiwUZk> | | |  | | --- | | **0. Equals() and hashcode()**    :    [**Link**](http://www.javaworld.com/article/2074996/hashcode-and-equals-method-in-java-object---a-pragmatic-concept.html) | |   HashCode and Equals method in Java object – A pragmatic concept By Debadatta Mishra  **Introduction**  Java.lang.Object has methods called hasCode() and equals(). These methods play a significant role in the real time application. However its use is not always common to all applications. In some case these methods are overridden to perform certain purpose. In this article I will explain you some concept of these methods and why it becomes necessary to override these methods.  **hashCode()**  As you know this method provides the has code of an object. Basically the default implementation of hashCode() provided by Object is derived by mapping the memory address to an integer value. If look into the source of Object class , you will find the following code for the hashCode. public native int hashCode(); It indicates that hashCode is the native implementation which provides the memory address to a certain extent. However it is possible to override the hashCode method in your implementation class.  **equals()**  This particular method is used to make equal comparison between two objects. There are two types of comparisons in Java. One is using “= =” operator and another is “equals()”. I hope that you know the difference between this two. More specifically the “.equals()” refers to equivalence relations. So in broad sense you say that two objects are equivalent they satisfy the “equals()” condition. If you look into the source code of Object class you will find the following code for the equals() method.  public boolean equals(Object obj)  {  return (this == obj);  }  Now I will explain you when to override the equals() and hashCode() methods and why it is necessary to override these methods. In this regard there is a rule of thumb that if you are going to override the one of the methods( ie equals() and hashCode() ) , you have to override the both otherwise it is a violation of contract made for equals() and hashCode(). Please refer to the Sun’s java docs for the method’s contract. I provide some test case scenario where you will find the significance of these methods. Case-1: You can override the hashCode method in your own way. Please refer to the following example.  package com.ddlab.core;  /\*\*  \* @author Debadatta Mishra(PIKU)  \*  \*/  public class Emp  {  private int age ;    public Emp( int age )  {  super();  this.age = age;  }    public int hashCode()  {  return age;  }  }  In the above example class “Emp” the variable age is the significant factor. Here the hashCode value will return the age of the person. Now let us consider the following test harness class.  package com.ddlab.core;  /\*\*  \* @author Debadatta Mishra(PIKU)  \*  \*/  public class TestEmp  {  public static void main(String[] args)  {  Emp emp1 = new Emp(23);  System.out.println("emp1.hashCode()--->>>"+emp1.hashCode());  }  }  If you run the above program, the output will be the age what you have given i.e. 23. Now question arises whether there is any way we can get the original hashCode(). We can say that if we do not override the hashCode() method what could have been the hashCode of this object. However please do not feel depressed, Java provide another approach even if you have overridden the hashCode() method , still you can get the original hashCode of a particular class. Now run the following test harness program.  package com.ddlab.core;  package com.ddlab.core;  /\*\*  \* @author Debadatta Mishra(PIKU)  \*  \*/  public class TestEmp  {  public static void main(String[] args)  {  Emp emp1 = new Emp(23);  System.out.println("Overridden hashCode()--->>>"+emp1.hashCode());  int originalHashCode = System.identityHashCode(emp1);  System.out.println("Original hashCode of Emp---->>>"+originalHashCode);  }  }  Here the output will be like this Overridden hashCode()--->>>23 Original hashCode of Emp---->>>8567361 As you know the above number is arbitrary, it depends upon your system. So then why it is necessary to override this method. There is one reason that if want to compare two objects based upon the equals() method. Although in a very simple class like “Emp”, you can achieve without overriding hashCode() method. But if you do this , you are going to violate the contract for the methods hashCode() and hashCode() of the object class. The similar case is for the method equals(). So funcational point is that if want to compare two objects based upon the equals() method you have to override both hashCode() and equals() methods. Please have look into the Emp class with the overridden methods and the related test harness class.  package com.ddlab.core;  /\*\*  \* @author Debadatta Mishra(PIKU)  \*  \*/  public class Emp  {  private int age ;    public Emp( int age )  {  super();  this.age = age;  }    public int hashCode()  {  return age;  }    public boolean equals( Object obj )  {  boolean flag = false;  Emp emp = ( Emp )obj;  if( emp.age == age )  flag = true;  return flag;  }  }  The related test harness class is given below.  package com.ddlab.core;  /\*\*  \* @author Debadatta Mishra(PIKU)  \*  \*/  public class TestEmp  {  public static void main(String[] args)  {  Emp emp1 = new Emp(23);  Emp emp2 = new Emp(23);  System.out.println("emp1.equals(emp2)--->>>"+emp1.equals(emp2));  }  }  Case- 2 Think of a test scenario where you want to store your objects in a HasSet and you want to find a particular object. First let us see if we do not override the methods and we want to store the objects in the HashSet. Let us analyse the impact of it from the following code.  package com.ddlab.core;  /\*\*  \* @author Debadatta Mishra(PIKU)  \*  \*/  public class Emp  {  private int age ;    public Emp( int age )  {  super();  this.age = age;  }    }  In the above code it is a normal class. Now let us see the test harness class.  package com.ddlab.core;  import java.util.HashSet;  /\*\*  \* @author Debadatta Mishra(PIKU)  \*  \*/  public class TestEmp  {  public static void main(String[] args)  {  Emp emp1 = new Emp(23);  Emp emp2 = new Emp(24);  Emp emp3 = new Emp(25);  Emp emp4 = new Emp(26);  Emp emp5 = new Emp(27);  HashSet<Emp> hs = new HashSet<Emp>();  hs.add(emp1);  hs.add(emp2);  hs.add(emp3);  hs.add(emp4);  hs.add(emp5);    System.out.println("HashSet Size--->>>"+hs.size());  System.out.println("hs.contains( new Emp(25))--->>>"+hs.contains(new Emp(25)));  System.out.println("hs.remove( new Emp(24)--->>>"+hs.remove( new Emp(24));  System.out.println("Now HashSet Size--->>>"+hs.size());  }  }  If you run the above program, the will output will be like the following. HashSet Size--->>>5 hs.contains( new Emp(25))--->>>false hs.remove( new Emp(24)--->>>false Now HashSet Size--->>>5 It means that you can not find the object. However it is not the case for Integer object. You can put object of type Integer in a HashSet and you can try and you can see the effect. Now let us modify the “Emp” class so that we will get over the problems what we faced in the above test harness class.  package com.ddlab.core;  /\*\*  \* @author Debadatta Mishra(PIKU)  \*  \*/  public class Emp  {  private int age ;    public Emp( int age )  {  super();  this.age = age;  }    public int hashCode()  {  return age;  }    public boolean equals( Object obj )  {  boolean flag = false;  Emp emp = ( Emp )obj;  if( emp.age == age )  flag = true;  return flag;  }  }  Here in the above class, we have overridden the hashCode() and equals() methods. Now if you run the same test harness class, you will get the desired output like the following. HashSet Size--->>>5 hs.contains( new Emp(25))--->>>true hs.remove( new Emp(24))--->>>true Now HashSet Size--->>>4 Case – 3 In this case you want to use your object as key not the value in the HashMap. So you have to override both the methods hashCode() and equals(). However it is left to the reader to create the object and test the feature in a Map. Case-4 If want to make your own immutable object , it will be wiser to override the equals() and hashCode() methods. To test the above programs, please create the appropriate package as mentioned in the program. You can also create your own package and modify the package name in the above programs. You can all the code in your favorable java editor.  **Conclusion**  I hope that you will enjoy my article. If you find any problems or errors, please feel free to send me a mail in the address  [debadattamishra@aol.com](mailto:debaattamishra@aol.com)  . This article is only meant for those who are new to java development. This article does not bear any commercial significance. Please provide me the feedback about this article.  **Chapter 12:Streams and Files**  In this tutorial we will learn how to read and write to files. We will see how to handle files.   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  | | --- | --- | | |  | | --- | | **0. Types of Streams :- Byte Streams, Character streams, Buffered Streams**    :    [**Link**](http://docs.oracle.com/javase/tutorial/essential/io/streams.html)  [**Link**](http://www.csie.ndhu.edu.tw/~showyang/NetProg2008/Assignment04.pdf) |  I/O Streams An I/O Stream represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.  Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways.  No matter how they work internally, all streams present the same simple model to programs that use them: A stream is a sequence of data. A program uses an *input stream* to read data from a source, one item at a time:  Reading information into a program.  Reading information into a program.  A program uses an *output stream* to write data to a destination, one item at time:  Writing information from a program.  Writing information from a program.  In this lesson, we'll see streams that can handle all kinds of data, from primitive values to advanced objects.  The data source and data destination pictured above can be anything that holds, generates, or consumes data. Obviously this includes disk files, but a source or destination can also be another program, a peripheral device, a network socket, or an array.  In the next section, we'll use the most basic kind of streams, byte streams, to demonstrate the common operations of Stream I/O. For sample input, we'll use the example file[xanadu.txt](http://docs.oracle.com/javase/tutorial/essential/io/examples/xanadu.txt), which contains the following verse:  In Xanadu did Kubla Khan  A stately pleasure-dome decree:  Where Alph, the sacred river, ran  Through caverns measureless to man  Down to a sunless sea. | | |  | | --- | | **0. Read and Write to files**    :    [**Link**](http://www.freejavaguide.com/file_streams.htm)  [**Link**](http://www.ntu.edu.sg/home/ehchua/programming/java/J2a_BasicsExercises.html) |   **File I/O and Streams**  You can write data to a file instead of the computer screen. You can write certain data to a file while still putting other data on the screen. Or you may need access to multiple files simultaneously. Or you may want to query the user for input rather than accepting it all on the command line. Or maybe you want to read data out of a file that's in a particular format. In Java all these methods take place as streams. < > Using File I/O *streams.* The System.out.println() statement we've been using all along is an implementation of Streams.  *A program that writes a string to a file*  In order to use the Java file classes, we must import the Java input/output package (java.io) in the following manner  import java.io.\*;  Inside the main method of our program, we must declare a FileOutputStream object. In this case, we wish to write a string to the file, and so we create a new PrintStream object that takes as its constructor the existing FileOutputStream. Any data we send from PrintStream will now be passed to the FileOutputStream, and ultimately to disk. We then make a call to the println method, passing it a string, and then close the connection.  Source Code  /\* \* FileOutput \* Demonstration of FileOutputStream and PrintStream classes \*/  import java.io.\*;  class FileOutput  {   public static void main(String args[]) {  FileOutputStream out; // declare a file output object PrintStream p; // declare a print stream object  try { // Create a new file output stream connected to "myfile.txt" out = new FileOutputStream("myfile.txt");  // Connect print stream to the output stream p = new PrintStream( out );  p.println ("This is written to a file myFile.txt");  p.close(); } catch (Exception e) { System.err.println ("Error writing to the file myFile.txt"); } } }  Interactively communicating with the user  *Program asking  the user for their name and then prints a personalized greeting.*  Source Code  import java.io.\*;  class PersonalHello {    public static void main (String args[])     {            byte name[] = new byte[100];       int nr\_read = 0;        System.out.println("Your name Please?");       try {         nr\_read = System.in.read(name);         System.out.print("Hello ");         System.out.write(name,0,nr\_read);       }       catch (IOException e) {         System.out.print("I did not get your name.");       }            }      }  In code that does any significant input or output you'll want to begin by importing all the various java.io classes. import.java.io.\*; Most of the reading and writing you do in Java will be done with bytes. Here we've started with an array of bytes that will hold the user's name.  First we print a query requesting the user's name. Then we read the user's name using the System.in.read() method. This method takes a byte array as an argument, and places whatever the user types in that byte array. Then, like before, we print "Hello." Finally we print the user's name.  The program doesn't actually see what the user types until he or she types a carriage return. This gives the user the chance to backspace over and delete any mistakes. Once the return key is pressed, everything in the line is placed in the array.  Reading Numbers  Often strings aren't enough. A lot of times you'll want to ask the user for a number as input. All user input comes in as strings so we need to convert the string into a number.  The getNextInteger() method that will accept an integer from the user. Here it is:    static int getNextInteger() {        String line;        DataInputStream in = new DataInputStream(System.in);     try {       line = in.readLine();       int i = Integer.valueOf(line).intValue();       return i;     }     catch (Exception e) {       return -1;     }           } // getNextInteger ends here  Reading Formatted Data  It's often the case that you want to read not just one number but multiple numbers. Sometimes you may need to read text and numbers on the same line. For this purpose Java provides the StreamTokenizer class.  Writing a text file  Sometimes you want to save your output in a  file. To do this we'll need to learn how to write data to a file.  Source Code  // Write the Fahrenheit to Celsius table in a file  import java.io.\*;  class FahrToCelsius  {    public static void main (String args[]) {      double fahr, celsius;     double lower, upper, step;      lower = 0.0;    // lower limit of temperature table     upper = 300.0;  // upper limit of temperature table     step  = 20.0;   // step size      fahr = lower;        try {        FileOutputStream fout =  new FileOutputStream("test.out");        // now to the FileOutputStream into a PrintStream       PrintStream myOutput = new PrintStream(fout);          while (fahr <= upper) {  // while loop begins here         celsius = 5.0 \* (fahr-32.0) / 9.0;         myOutput.println(fahr + " " + celsius);         fahr = fahr + step;       } // while loop ends here        }  // try ends here     catch (IOException e) {       System.out.println("Error: " + e);       System.exit(1);     }      } // main ends here  }  There are only three things necessary to write formatted output to a file rather than to the standard output:   1. Open a *FileOutputStream* using a line like   FileOutputStream fout =  new FileOutputStream("test.out");  This line initializes the FileOutputStream with the name of the file you want to write into.   1. Convert the FileOutputStream into a *PrintStream* using a statement like   PrintStream myOutput = new PrintStream(fout);  The PrintStream is passed the FileOutputStream from step 1.   1. Instead of using System.out.println() use myOutput.println(). System.out and myOutput are just different instances of the PrintStream class. To print to a differentPrintStream we keep the syntax the same but change the name of the PrintStream.   Reading a text file  Now that we know how to write a text file, let's try reading one. The following code accepts a series of file names on the command line and then prints those filenames to the standard output in the order they were listed.  // Imitate the Unix cat utility  import java.io.\*;  class cat  {    public static void main (String args[]) {      String thisLine;    //Loop across the arguments   for (int i=0; i < args.length; i++) {     //Open the file for reading   try {     FileInputStream fin =  new FileInputStream(args[i]);      // now turn the FileInputStream into a DataInputStream     try {       DataInputStream myInput = new DataInputStream(fin);          try {         while ((thisLine = myInput.readLine()) != null) {  // while loop begins here           System.out.println(thisLine);         } // while loop ends here       }       catch (Exception e) {        System.out.println("Error: " + e);       }     } // end try     catch (Exception e) {       System.out.println("Error: " + e);     }       } // end try    catch (Exception e) {     System.out.println("failed to open file " + args[i]);     System.out.println("Error: " + e);   }   } // for end here      } // main ends here  } Java Programming TutorialExercises on Java Basics1.  Exercises on Flow Controls1.1  Exercises on Conditional (Decision) **Exercise CheckPassFail (if-else):** Write a program called **CheckPassFail** which prints "PASS" if the int variable "mark" is more than or equal to 50; or prints "FAIL" otherwise.  Hints:  public class **CheckPassFail** { // saved as "CheckPassFail.java"  public static void main(String[] args) {  int mark = 49; // set the value of mark here!  System.out.println("The mark is " + mark);    if ( ...... ) {  System.out.println( ...... );  } else {  System.out.println( ...... );  }  }  }  **Exercise CheckOddEven (if-else):** Write a program called **CheckOddEven** which prints "Odd Number" if the int variable “number” is odd, or “Even Number” otherwise.  Hints: n is an even number if (n % 2) is 0.  public class **CheckOddEven** { // saved as "CheckOddEven.java"  public static void main(String[] args) {  int number = 49; // set the value of number here!  System.out.println("The number is " + number);  if ( ...... ) {  System.out.println( ...... );  } else {  System.out.println( ...... );  }  }  }  **Exercise PrintNumberInWord (nested-if, switch-case):** Write a program called **PrintNumberInWord** which prints "ONE", "TWO",... , "NINE", "OTHER" if the int variable "number" is 1, 2,... , 9, or other, respectively. Use (a) a "nested-if" statement; (b) a "switch-case" statement.  Hints:  public class **PrintNumberInWord** { // saved as "PrintNumberInWord.java"  public static void main(String[] args) {  int number = 5;    // Using nested-if  if (number == 1) {  System.out.println("ONE");  } else if (......) {  ......  } else if (......) {  ......  ......  } else {  ......  }    // Using switch-case  switch(number) {  case 1: System.out.println("ONE"); break;  case 2: ......  ......  ......  default: System.out.println("OTHER");  }  }  }  Similarly, write a program called **PrintDayInWord**, which prints “Sunday”, “Monday”, ... “Saturday” if the int variable "day" is 0, 1, ..., 6, respectively.  Otherwise, it shall print “Not a valid day”. 1.2  Exercises on Loop (Iteration) **Exercise SumAndAverage (Loop):** Write a program called **SumAndAverage** to produce the sum of 1, 2, 3, ..., to an upperbound (e.g., 100). Also compute and display the average. The output shall look like:  The sum is 5050  The average is 50.5  Hints:  public class **SumAndAverage** { // saved as "SumAndAverage.java"  public static void main (String[] args) {  int sum = 0; // store the accumulated sum, init to 0  double average; // average in double  int lowerbound = 1; // the lower bound to sum  int upperbound = 100; // the upper bound to sum  for (int number = lowerbound; number <= upperbound; ++number) { // for loop  sum += number; // same as "sum = sum + number"  }  // Compute average in double. Beware that int/int produces int.  ......  // Print sum and average.  ......  }  }  TRY:   1. Modify the program to use a "while-do" loop instead of "for" loop. 2. int number = lowerbound; 3. int sum = 0; 4. while (number <= upperbound) { 5. sum += number; 6. ++number;   }   1. Modify the program to use a "do-while" loop. 2. int number = lowerbound; 3. int sum = 0; 4. do { 5. sum += number; 6. ++number;   } while (number <= upperbound);   1. What is the difference between "for" and "while-do" loops? What is the difference between "while-do" and "do-while" loops? 2. Modify the program to sum from 111 to 8899, and compute the average. Introduce an int variable called count to count the numbers in the specified range. 3. int count = 0; // count the number within the range, init to 0 4. for (...; ...; ...) { 5. ...... 6. ++count;   }   1. Modify the program to sum only the odd numbers from 1 to 100, and compute the average. (Hint: n is an odd number if n % 2 is not 0.) 2. Modify the program to sum those numbers from 1 to 100 that is divisible by 7, and compute the average. 3. Modify the program to find the "sum of the squares" of all the numbers from 1 to 100, i.e. 1\*1 + 2\*2 + 3\*3 + ... + 100\*100.   **Exercise Product1ToN (Loop):** Write a program called **Product1ToN** to compute the product of integers 1 to 10 (i.e., 1×2×3×...×10). Try computing the product from 1 to 11, 1 to 12, 1 to 13 and 1 to 14. Write down the product obtained and explain the results.  Hints: Declares an int variable called product (to accumulate the product) and initialize to 1.  **Exercise HarmonicSum (Loop):** Write a program called **HarmonicSum** to compute the sum of a harmonic series, as shown below, where n=50000. The program shall compute the sum from left-to-right as well as from the right-to-left. Obtain the difference between these two sums and explain the difference. Which sum is more accurate?  ExerciseBasics_HarmonicSum.png  Hints:  public class **HarmonicSum** { // saved as "HarmonicSum.java"  public static void main (String[] args) {  int maxDenominator = 50000;  double sumL2R = 0.0; // sum from left-to-right  double sumR2L = 0.0; // sum from right-to-left    // for-loop for summing from left-to-right  for (int denominator = 1; denominator <= maxDenominator; ++denominator) {  ......  // Beware that int/int gives int.  }  // for-loop for summing from right-to-left  ......  // Find the difference and display  ......  }  }  **Exercise ComputePI (Loop & Condition):** Write a program called **ComputePI** to compute the value of π, using the following series expansion. You have to decide on the termination criterion used in the computation (such as the number of terms used or the magnitude of an additional term). Is this series suitable for computing π?  ExerciseBasics_ComputePI.png  JDK maintains the value of π in a double constant called Math.PI. Compare the values obtained and the Math.PI, in percents of Math.PI.  Hint: Add to sum if the denominator modulus 4 is 1, and subtract from sum if it is 3.  double sum = 0;  int maxDenom = 10000000;  for (int denom = 1; ..... ; denom = denom + 2) {  if (denom % 4 == 1) {  sum += ......;  } else if (denom % 4 == 3) {  sum -= ......;  } else {  System.out.println("The computer has gone crazy?!");  }  }  **Exercise CozaLozaWoza (Loop & Condition):** Write a program called **CozaLozaWoza** which prints the numbers 1 to 110, 11 numbers per line. The program shall print "Coza" in place of the numbers which are multiples of 3, "Loza" for multiples of 5, "Woza" for multiples of 7, "CozaLoza" for multiples of 3 and 5, and so on. The output shall look like:  1 2 Coza 4 Loza Coza Woza 8 Coza Loza 11  Coza 13 Woza CozaLoza 16 17 Coza 19 Loza CozaWoza 22  23 Coza Loza 26 Coza Woza 29 CozaLoza 31 32 Coza  ......  Hints:  public class **CozaLozaWoza** { // saved as "CozaLozaWoza.java"  public static void main(String[] args) {  int lowerbound = 1;  int upperbound = 110;  for (int number = lowerbound; number <= upperbound; ++number) {  // Print "Coza" if number is divisible by 3  if (......) {  System.out.print("Coza");  }  // Print "Loza" if number is divisible by 5  if (......) {  System.out.print(.....);  }  // Print "Woza" if number is divisible by 7  ......  // Print the number if it is not divisible by 3, 5 and 7  if (......) {  ......  }  // Print a newline if number is divisible by 11; else print a space  if (......) {  System.out.println();  }  }  }  }  TRY: Modify the program to use nested-if (if ... else if ... else if ... else) instead.  **Exercise Fibonacci (Loop):** Write a program called **Fibonacci** to display the first 20 Fibonacci numbers F(n), where F(n)=F(n–1)+F(n–2) and F(1)=F(2)=1. Also compute their average. The output shall look like:  The first 20 Fibonacci numbers are:  1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765  The average is 885.5  Hints:  public class **Fibonacci** {  public static void main (String args[]) {  int n = 3; // the index n for F(n), starting from n=3  int fn; // F(n) to be computed  int fnMinus1 = 1; // F(n-1), init to F(2)  int fnMinus2 = 1; // F(n-2), init to F(1)  int nMax = 20; // maximum n, inclusive  int sum = fnMinus1 + fnMinus2;  double average;    System.out.println("The first " + nMax + " Fibonacci numbers are:");  ......    while (n <= nMax) {  // Compute F(n), print it and add to sum  ......  // Adjust the index n and shift the numbers  ......  }    // Compute and display the average (=sum/nMax)  ......  }  }  Tribonacci numbers are a sequence of numbers T(n) similar to Fibonacci numbers, except that a number is formed by adding the three previous numbers, i.e., T(n)=T(n-1)+T(n-2)+T(n-3),T(1)=T(2)=1, and T(3)=2. Write a program called **Tribonacci** to produce the first twenty Tribonacci numbers.  **Extracting Digits**: Write a program to extract each digit from an int, in the reverse order. For example, if the int is 1542, the output shall be "2,4,5,1", with a comma separating the digits.  Hints: Use n % 10 to extract a digit; and n = n / 10 to discard the last digit.  int n = ....;  while (n > 0) {  int digit = n % 10; // Extract the last digit  ......  .....  n = n / 10; // Drop last digit and repeat the loop  } 1.3  Exercises on Nested-Loop **Exercise SquareBoard (nested-loop):** Write a program called **SquareBoard** that displays the following n×n (n=5) pattern using two nested for-loops.  # # # # #  # # # # #  # # # # #  # # # # #  # # # # #  Your program should use only two output statements, one EACH of the followings:  System.out.print("# "); // print # and a space, without newline  System.out.println(); // print a newline  Hints:  public class **SquareBoard** { // saved as "SquareBoard.java"  public static void main (String[] args) {  int size = 5; // size of the board  for (int row = 1; ......; ......) {  for (int col = 1; ......; ......) {  ......  }  ......  }  }  }  **Exercise CheckerBoard (nested-loop):** Write a program called **CheckerBoard** that displays the following n×n (n=7) checkerboard pattern using two nested for-loops.  # # # # # # #  # # # # # # #  # # # # # # #  # # # # # # #  # # # # # # #  # # # # # # #  # # # # # # #  Your program should use only three output statements, one EACH of the followings:  System.out.print("# "); // print # and a space, without newline  System.out.print(" "); // print a space, without newline  System.out.println(); // print a newline  Hints:  public class **CheckerBoard** { // saved as "CheckerBoard.java"  public static void main (String[] args) {  int size = 7; // size of the board  for (int row = 1; ......; ......) {  // Use modulus 2 to find alternate lines  if ((row % 2) == 0) { // row 2, 4, 6, 8  ......  }  for (int col = 1; ......; ......) {  ......  }  ......  }  }  }  **Exercise TimeTable (nested-loop):** Write a program called TimeTable to produce the multiplication table of 1 to 9 as shown using two nested for-loops:  \* | 1 2 3 4 5 6 7 8 9  -------------------------------  1 | 1 2 3 4 5 6 7 8 9  2 | 2 4 6 8 10 12 14 16 18  3 | 3 6 9 12 15 18 21 24 27  4 | 4 8 12 16 20 24 28 32 36  5 | 5 10 15 20 25 30 35 40 45  6 | 6 12 18 24 30 36 42 48 54  7 | 7 14 21 28 35 42 49 56 63  8 | 8 16 24 32 40 48 56 64 72  9 | 9 18 27 36 45 54 63 72 81  Modify the program to print the multiplication table of 1 to 12. 2.  Exercises on Keyboard and File Input **Exercise KeyboardScanner (Keyboard Input):** Write a program called KeyboardScanner to prompt user for an int, a double, and a String. The output shall look like (the inputs are shown in bold):  Enter an integer: **12**  Enter a floating point number: **33.44**  Enter your name: **Peter**  Hi! Peter, the sum of 12 and 33.44 is 45.44  Hints:  import java.util.Scanner; // needed to use Scanner for input  public class **KeyboardScanner** {  public static void main(String[] args) {  int num1;  double num2;  String name;  double sum;    // Setup a Scanner called in to scan the keyboard (System.in)  Scanner in = new Scanner(System.in);  System.out.print("Enter an integer: ");  num1 = in.nextInt(); // use nextInt() to read int  System.out.print("Enter a floating point number: ");  num2 = in.nextDouble(); // use nextDouble() to read double  System.out.print("Enter your name: ");  name = in.next(); // use next() to read String    // Display  ......  }  }  **Exercise FileScanner (File Input):** Write a program called **FileScanner** to read an int, a double, and a String form a text file called "in.txt", and produce the following output:  The integer read is 12  The floating point number read is 33.44  The String read is "Peter"  Hi! Peter, the sum of 12 and 33.44 is 45.44  You need to create a text file called "in.txt" (in Eclipse, right-click on the "project" ⇒ "New" ⇒ "File") with the following contents:  12  33.44  Peter  import java.util.Scanner; // Needed to use Scanner for input  import java.io.File; // Needed to use File  import java.io.FileNotFoundException; // Needed for file operation    public class **FileScanner** {  public static void main(String[] args)  throws FileNotFoundException { // Needed for file operation  int num1;  double num2;  String name;  double sum;    // Setup a Scanner to read from a text file  Scanner in = new Scanner(new File("in.txt"));  num1 = in.nextInt(); // use nextInt() to read int  num2 = in.nextDouble(); // use nextDouble() to read double  name = in.next(); // use next() to read String    // Display  ......  }  }  **Exercise CircleComputation (User Input):** Write a program called **CircleComputation**, which prompts user for a radius (of double) and compute the area and perimeter of a circle. The output shall look like:  Enter the radius: **1.2**  The area is 4.5239  The perimeter is 7.5398223686155035  Hints: π is kept in a constant called Math.PI. 3.  Exercises on User Input and String Operations **Exercise ReverseString:** Write a program called ReverseString, which prompts user for a String, and prints the reverse of the String. The output shall look like:  Enter a String: **abcdef**  The reverse of String "abcdef" is "fedcba".  Hints:  import java.util.Scanner;  public class **ReverseString** {  public static void main(String[] args) {  String inStr; // input String  int inStrLen; // length of the input String    Scanner in = new Scanner(System.in);  System.out.print("Enter a String: ");  inStr = in.next(); // use next() to read String  inStrLen = inStr.length();    // Use inStr.charAt(index) to extract character at 'index' from inStr  ......  }  }  For a String called inStr, you can use inStr.length() to get the length of the String; and inStr.charAt(index) to retrieve the char at the index position, where index begins with 0.  **Exercise PhoneKeyPad:** On your phone keypad, the alphabets are mapped to digits as follows: ABC(2), DEF(3), GHI(4), JKL(5), MNO(6), PQRS(7), TUV(8), WXYZ(9).  Write a program called PhoneKeyPad, which prompts user for a String (case insensitive), and converts to a sequence of Keypad digits. Use a nested-if (or switch-case) in this exercise. Modify your program to use an array for table look-up later.  Hints: You can use in.next().toLowerCase() to read a string and convert it to lowercase to reduce your cases.  **Exercise TestPalindromicWord:** A word that reads the same backward as forward is called a palindrome, e.g., "mom", "dad", "racecar", "madam", and "Radar" (case-insensitive). Write a program called TestPalindromicWord, that prompts user for a word and prints ""xxx" is|is not a palindrome".  Hints: Read in a word and convert to lowercase via in.next().toLowercase().  A phrase that reads the same backward as forward is also called a palindrome, e.g., "Madam, I'm Adam", "A man, a plan, a canal - Panama!" (ignoring punctuation and capitalization). Modify your program (called TestPalindromicPhrase) to test palindromic phrase.  Hints: Read in the lowercase phrase via in.nextLine().toLowercase(). Maintain two indexes, forwardIndex and backwardIndex, used to scan the phrase forward and backward.  **Exercise Bin2Dec:** Write a program called Bin2Dec to convert an input binary string into its equivalent decimal number. Your output shall look like:  Enter a Binary string: **1011**  The equivalent decimal number for binary "1011" is 11    Enter a Binary string: **1234**  Error: Invalid Binary String "1234"  Hints: For a n-bit binary number bn-1bn-2...b1b0, bi∈{0,1}, the equivalent decimal number is bn-1×2n-1+bn-2×2n-2+ ...+b1×21+b0×20.  import java.util.Scanner;  public class **Bin2Dec** {  public static void main(String[] args) {  String binStr; // input binary string  int binStrLen; // length of the input string  int dec = 0; // equivalent decimal number  char binChar; // each individual char in the binary string    Scanner in = new Scanner(System.in);    // Read input binary string  ......    // Convert binary string into Decimal  ......  }  }  binStr : 1 0 1 1 1 0 0 1  charAt(idx) : 0 1 2 3 4 5 6 7  Math.pow(2, order) : 7 6 5 4 3 2 1 0    binStr.length() = 8  idx + order = binStr.length() - 1  You can use JDK method Math.pow(x, y) to compute the x raises to the power of y. This method takes two doubles as argument and returns a double. You may have to cast the result back to int.  To convert a char (of digit '0' to '9') to int (0 to 9), simply subtract by char '0', e.g., '5'-'0' gives int 5.  **Exercise Hex2Dec:** Write a program called Hex2Dec to convert an input hexadecimal string into its equivalent decimal number. Your output shall look like:  Enter a Hexadecimal string: **1a**  The equivalent decimal number for hexadecimal "1a" is 26    Enter a Hexadecimal string: **1y3**  Error: Invalid Hexadecimal String "1y3"  Hints:  For a n-digit hexadecimal number hn-1hn-2...h1h0, hi∈{0,…,9,A,…,F}, the equivalent decimal number is hn-1×16n-1+hn-2×16n-2+ ...+h1×161+h0×160.  You do not need a big nested-if statement of 16 cases (or 22 considering the upper and lower letters). Extract the individual character from the hexadecimal string, says c. If char c is between'0' to '9', you can get the integer offset via c-'0'. If c is between 'a' to 'f' or 'A' to 'F', the integer offset is c-'a'+10 or c-'A'+10.  String hexStr;  char hexChar;  ......  hexChar = hexStr.charAt(i);  ......  if (hexChar >= '0' && hexChar <= '9') {  ... (hexChar-'0') ...  ...  } else if (hexChar >= 'a' && hexChar <= 'f') { // lowercase  ... (hexChar-'a'+10) ...  ...  } else if (hexChar >= 'A' && hexChar <= 'F') { // uppercase  ... (hexChar-'A'+10) ...  ...  } else {  System.out.println("Error: Invalid hexadecimal string");  System.exit(1); // quit the program  } 4.  Exercises on Array **Exercise GradesAverage (Array):** Write a program called GradesAverage, which prompts user for the number of students, reads it from the keyboard, and saves it in an int variable called numStudents. It then prompts user for the grades of each of the students and saves them in an int array called grades.  Your program shall check that the grade is between 0 and 100. A sample session is as follow:  Enter the number of students: **3**  Enter the grade for student 1: **55**  Enter the grade for student 2: **108**  Invalid grade, try again...  Enter the grade for student 2: **56**  Enter the grade for student 3: **57**  The average is 56.0  **Exercise Hex2Bin (Array and Table Lookup):** Write a program called Hex2Bin to convert a hexadecimal string into its equivalent binary string. The output shall look like:  Enter a Hexadecimal string: **1abc**  The equivalent binary for hexadecimal "1abc" is 0001 1010 1011 1100  Hints: Use an array of 16 binary Strings corresponding to hexadecimal number '0' to 'F' (or 'f'), as follows:  String[] hexBits = {"0000", "0001", "0010", "0011",  "0100", "0101", "0110", "0111",  "1000", "1001", "1010", "1011",  "1100", "1101", "1110", "1111"}; 5.  Exercises on Command-line Arguments **Exercise Arithmetic (Command-line arguments):** Write a program called **Arithmetic** that takes three command-line arguments: two integers followed by an arithmetic operator (+, -, \* or /). The program shall perform the corresponding operation on the two integers and print the result. For example:  > java Arithmetic **3 2 +**  3+2=5    > java Arithmetic **3 2 -**  3-2=1    > java Arithmetic **3 2 /**  3/2=1  Hints:  The method main(String[] args) takes an argument: "an array of String", which is often (but not necessary) named args. This parameter captures the command-line arguments supplied by the user when the program is invoked. For example, if a user invokes:  > java Arithmetic **12345 4567 +**  The three command-line arguments "12345", "4567" and "+" will be captured in a String array {"12345", "4567", "+"} and passed into the main() method as the argument args. That is,  args is {"12345", "4567", "+"}; // args is a String array  args.length is 3; // length of the array  args[0] is "12345"; // 1st element of the String array  args[1] is "4567"; // 2nd element of the String array  args[2] is "+"; // 3rd element of the String array  args[0].length() is 5; // length of 1st String element  args[1].length() is 4; // length of the 2nd String element  args[2].length() is 1; // length of the 3rd String element  public class **Arithmetic** {  public static void main (String[] args) {  int operand1, operand2;  char theOperator;    // Check if there are 3 command-line arguments in the  // String array args[] by using length variable of array.  if (args.length != 3) {  System.err.println("Usage: java Arithmetic int1 int2 op");  return;  }    // Convert the 3 Strings args[0], args[1], args[2] to int and char.  // Use the Integer.parseInt(aStr) to convert a String to an int.  operand1 = Integer.parseInt(args[0]);  operand2 = ......    // Get the operator, assumed to be the first character of  // the 3rd string. Use method charAt() of String.  theOperator = args[2].charAt(0);  System.out.print(args[0] + args[2] + args[1] + "=");    switch(theOperator) {  case ('-'): System.out.println(operand1 - operand2); break;  case ('+'): ......  case ('\*'): ......  case ('/'): ......  default:  System.err.println("Error: invalid operator!");  }  }  }  Notes:   * To provide command-line arguments, use the "cmd" shell to run your program in the form "java ClassName arg1 arg2 ....". * To provide command-line arguments in Eclipse, right click the source code ⇒ "Run As" ⇒ "Run Configurations..." ⇒ Select "Main" and choose the proper main class ⇒ Select "Arguments" ⇒ Enter the command-line arguments, e.g., "3 2 +" in "Program Arguments". * To provide command-line arguments in NetBeans, right click the "Project" name ⇒ "Set Configuration" ⇒ "Customize..." ⇒ Select categories "Run" ⇒ Enter the command-line arguments, e.g., "3 2 +" in the "Arguments" box (but make sure you select the proper Main class).   Question: Try "java Arithmetic 2 4 \*" (in CMD shell and Eclipse/NetBeans) and explain the result obtained. How to resolve this problem?  In Windows' CMD shell, \* is known as a wildcard character, that expands to give the list of file in the directory (called Shell Expansion). For example, "dir \*.java" lists all the file with extension of ".java". You could double-quote the \* to prevent shell expansion. Eclipse has a bug in handling this, even \* is double-quoted. NetBeans??  **Exercise SumDigits (Command-line arguments):** Write a program called SumDigits to sum up the individual digits of a positive integer, given in the command line. The output shall look like:  > **java SumDigits 12345**  The sum of digits = 1 + 2 + 3 + 4 + 5 = 15 6.  Exercises on Method **Exercise GradesStatistics (Method):** Write a program called **GradesStatistics**, which reads in n grades (of int between 0 and 100, inclusive) and displays the average, minimum,maximum, and standard deviation. Your program shall check for valid input. You should keep the grades in an int[] and use a method for each of the computations. Your output shall look like:  Enter the number of students: **4**  Enter the grade for student 1: **50**  Enter the grade for student 2: **51**  Enter the grade for student 3: **56**  Enter the grade for student 4: **53**  The average is 52.5  The minimum is 50  The maximum is 56  The standard deviation is 2.29128784747792  Hints: The formula for calculating standard deviation is:  ExerciseBasics_GradesAverage.png  public class **GradesStatistics** {  public static int[] grades; // Declare an int[], to be allocated later    // main() method  public static void main(String[] args) {  readGrades();  System.out.println("The average is " + average());  System.out.println("The minimum is " + min());  System.out.println("The maximum is " + max());  System.out.println("The standard deviation is " + stdDev());  }    // Prompt user for the number of students and allocate the "grades" array.  // Then, prompt user for grade, check for valid grade, and store in "grades".  public static void readGrades() { ....... }    // Return the average value of int[] grades  public static double average() { ...... }    // Return the maximum value of int[] grades  public static int max() { ...... }    // Return the minimum value of int[] grades  public static int min() { ....... }    // Return the standard deviation of the int[] grades  public static double stdDev() { ....... }  }  **Exercise GradesHistogram (Method):** Write a program called **GradesHistogram**, which reads in n grades (of int between 0 and 100, inclusive) from a text file called "grades.in" and displays the histogram horizontally and vertically. The file has the following format:  numStduents:int  grade1:int grade2:int .... gradeN:int  For example:  15  49 50 51 59 0 5 9 10 15 19 50 55 89 99 100  The output shall consist of a horizontal histogram and a vertical histogram as follows:  0 - 9: \*\*\*  10 - 19: \*\*\*  20 - 29:  30 - 39:  40 - 49: \*  50 - 59: \*\*\*\*\*  60 - 69:  70 - 79:  80 - 89: \*  90 -100: \*\*  \*  \*  \* \* \*  \* \* \* \*  \* \* \* \* \* \*  0-9 10-19 20-29 30-39 40-49 50-59 60-69 70-79 80-89 90-100  Hints:  public class **GradesHistogram** {  public static int[] grades;  // Declare an int array of grades, to be allocated later  public static int[] bins = new int[10];  // Declare and allocate an int array for histogram bins.  // 10 bins for 0-9, 10-19,...., 90-100    public static void main(String[] args) {  readGrades("grades.in");  computeHistogram();  printHistogramHorizontal();  printHistogramVertical();  }    // Read the grades from "filename", store in "grades" array.  // Assume that the inputs are valid.  public static void readGrades(String filename) { ...... }  // Based on "grades" array, populate the "bins" array.  public static void computeHistogram() { ....... }  // Print histogram based on the "bins" array.  public static void printHistogramHorizontal() { ...... }  // Print histogram based on the "bins" array.  public static void printHistogramVertical() { ...... }  }  **Exercise ReverseArrayTest (Method):** Write a method called reverseArray() with the following signature:  public static void reverseArray(int[] intArray)  The method accepts an int array, and reverses its orders. For example, if the input array is {12, 56, 34, 79, 26}, the reversal is {26, 79, 34, 56, 12}. You MUST NOT use another array in your method (but you need a temporary variable to do the swap). Also write a test class called ReverseArrayTest to test this method.  Take note that the array passed into the method can be modified by the method (this is called "pass by reference"). On the other hand, primitives passed into a method cannot be modified. This is because a clone is created and passed into the method instead of the original copy (this is called "pass by value"). 7.  More (Difficult) Exercises **Exercise (JDK Source Code):** Extract the source code of the class Math from the JDK source code ("$JAVA\_HOME" ⇒ "src.zip" ⇒ "Math.java" under folder "java.lang"). Study how constants such as E and PI are defined. Also study how methods such as abs(), max(), min(), toDegree(), etc, are written.  **Exercise Matrix:** Similar to Math class, write a Matrix library that supports matrix operations (such as addition, subtraction, multiplication) via 2D arrays. The operations shall support bothdoubles and ints. Also write a test class to exercise all the operations programmed.  Hints:  public class **Matrix** {  public static void printMatrix(int[][] m) { ...... }  public static void printMatrix(double[][] m) { ...... }  public static boolean haveSameDimension(int[][] m1, int[][] m2) { ...... }  public static boolean haveSameDimension(double[][] m1, double[][] m2) { ...... }  public static int[][] add(int[][] m1, int[][] m2) { ...... }  public static double[][] add(double[][] m1, double[][] m2) { ...... }  ......  }  **Exercise PrintAnimalPattern (Special Characters and Escape Sequences):** Write a program called **PrintAnimalPattern**, which uses println() to produce this pattern:  '\_\_'  (©©)  /========\/  / || %% ||  \* ||----||  ¥¥ ¥¥  "" ""  Hints:   * Use escape sequence \uhhhh where hhhh are four hex digits to display Unicode characters such as ¥ and ©. ¥ is 165 (00A5H) and © is 169 (00A9H) in both ISO-8859-1 (Latin-1) and Unicode character sets. * Double-quote (") and black-slash (\) require escape sign inside a String. Single quote (') does not require escape sign.   TRY: Print the same pattern using printf(). (Hints: Need to use %% to print a % in printf() because % is the suffix for format specifier.)  **Exercise PrintPatterns:** Write a method to print each of the followings patterns using nested loops in a class called **PrintPatterns**. The signatures of the methods are:  public static void printPatternX(int size) // 'X' from 'A' to ..., size is a positive integer.  # # # # # # # # # # # # # # # # # #  # # # # # # # # # # # # # # # # # #  # # # # # # # # # # # # # # # # # #  # # # # # # # # # # # # # # # # # #  # # # # # # # # # # # # # # # # # #  # # # # # # # # # # # # # # # # # #  # # # # # # # # # # # # # # # # # #  # # # # # # # # # # # # # # # # # #  (a) (b) (c) (d)    Hints: On the diagonal, row = col. On the opposite diagonal, row + col = size + 1.    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #  # # # # # # # # # #  # # # # # # # # # #  # # # # # # # #  # # # # # # # # # #  # # # # # # # # # #  # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #  (e) (f) (g) (h) (i)    # # # # # # # # # # # # #  # # # # # # # # # # # # # # #  # # # # # # # # # # # # # # # # #  # # # # # # # # # # # # # # # # # # #  # # # # # # # # # # # # # # # # # # # # #  # # # # # # # # # # # # # # # # # # # # # # #  (j) (k) # # # # # # # # #  # # # # # # #  # # # # #  # # #  #  (l)    1 1 2 3 4 5 6 7 8 1 8 7 6 5 4 3 2 1  1 2 1 2 3 4 5 6 7 2 1 7 6 5 4 3 2 1  1 2 3 1 2 3 4 5 6 3 2 1 6 5 4 3 2 1  1 2 3 4 1 2 3 4 5 4 3 2 1 5 4 3 2 1  1 2 3 4 5 1 2 3 4 5 4 3 2 1 4 3 2 1  1 2 3 4 5 6 1 2 3 6 5 4 3 2 1 3 2 1  1 2 3 4 5 6 7 1 2 7 6 5 4 3 2 1 2 1  1 2 3 4 5 6 7 8 1 8 7 6 5 4 3 2 1 1  (m) (n) (o) (p)    1 1 2 3 4 5 6 7 8 7 6 5 4 3 2 1  1 2 1 1 2 3 4 5 6 7 6 5 4 3 2 1  1 2 3 2 1 1 2 3 4 5 6 5 4 3 2 1  1 2 3 4 3 2 1 1 2 3 4 5 4 3 2 1  1 2 3 4 5 4 3 2 1 1 2 3 4 3 2 1  1 2 3 4 5 6 5 4 3 2 1 1 2 3 2 1  1 2 3 4 5 6 7 6 5 4 3 2 1 1 2 1  1 2 3 4 5 6 7 8 7 6 5 4 3 2 1 1  (q) (r)    1 1 1 2 3 4 5 6 7 8 7 6 5 4 3 2 1  1 2 2 1 1 2 3 4 5 6 7 7 6 5 4 3 2 1  1 2 3 3 2 1 1 2 3 4 5 6 6 5 4 3 2 1  1 2 3 4 4 3 2 1 1 2 3 4 5 5 4 3 2 1  1 2 3 4 5 5 4 3 2 1 1 2 3 4 4 3 2 1  1 2 3 4 5 6 6 5 4 3 2 1 1 2 3 3 2 1  1 2 3 4 5 6 7 7 6 5 4 3 2 1 1 2 2 1  1 2 3 4 5 6 7 8 7 6 5 4 3 2 1 1 1  (s) (t)    1  2 3 2  3 4 5 4 3  4 5 6 7 6 5 4  5 6 7 8 9 8 7 6 5  6 7 8 9 0 1 0 9 8 7 6  7 8 9 0 1 2 3 2 1 0 9 8 7  8 9 0 1 2 3 4 5 4 3 2 1 0 9 8  (u)  **Exercise PrintTriangles:** Write a method to print each of the following patterns using nested-loops in a class called **PrintTriangles**. The signatures of the methods are:  public static void printXxxTriangle(int numRows) // Xxx is the pattern's name  Write the main() which prompts the user for the numRows and prints all the patterns.  1  1 2 1  1 2 4 2 1  1 2 4 8 4 2 1  1 2 4 8 16 8 4 2 1  1 2 4 8 16 32 16 8 4 2 1  1 2 4 8 16 32 64 32 16 8 4 2 1  1 2 4 8 16 32 64 128 64 32 16 8 4 2 1  (a) PowerOf2Triangle    1 1  1 1 1 1  1 2 1 1 2 1  1 3 3 1 1 3 3 1  1 4 6 4 1 1 4 6 4 1  1 5 10 10 5 1 1 5 10 10 5 1  1 6 15 20 15 6 1 1 6 15 20 15 6 1  (b) PascalTriangle1 (c) PascalTriangle2  **Exercise TrigonometricSeries:** Write a method to compute sin(x) and cos(x) using the following series expansion, in a class called **TrigonometricSeries**. The headers of the methods are:  public static double sin(double x, int numTerms) // x in radians  public static double cos(double x, int numTerms)  ExerciseBasics_TrigonometricSeries.png  Compare the values computed using the series with the JDK methods Math.sin(), Math.cos() at x=0, π/6, π/4, π/3, π/2 using various numbers of terms.  Hints: Avoid generating large numerator and denominator (which may cause arithmetic overflow, e.g., 13! is out of int range). Compute the terms as:  ExerciseBasics_TrigonometricSeriesHint.png  **Exercise SpecialSeries:** Write a method to compute the sum of the series in a class called SpecialSeries. The signature of the method is:  public static double sumOfSeries(double x, int numTerms)  ExerciseBasics_Series.png  **Exercise FibonacciInt (Overflow) :** Write a program called **FibonacciInt** to list all the Fibonacci numbers, which can be expressed as an int (i.e., 32-bit signed integer in the range of [-2147483648, 2147483647]). The output shall look like:  F(0) = 1  F(1) = 1  F(2) = 2  ...  F(45) = 1836311903  F(46) is out of the range of int  Hints: The maximum and minimum values of a 32-bit int are kept in constants Integer.MAX\_VALUE and Integer.MIN\_VALUE, respectively. Try these statements:  System.out.println(Integer.MAX\_VALUE);  System.out.println(Integer.MIN\_VALUE);  System.out.println(Integer.MAX\_VALUE + 1);  Take note that in the third statement, Java Runtime does not flag out an overflow error, but silently wraps the number around. Hence, you cannot use F(n-1) + F(n-2) > Integer.MAX\_VALUE to check for overflow. Instead, overflow occurs for F(n) if (Integer.MAX\_VALUE – F(n-1)) < F(n-2) (i.e., no room for the next Fibonacci number).  Write a similar program for Tribonacci numbers.  **Exercise FactorialInt (Overflow):** Write a program called **Factorial1to10**, to compute the factorial of n, for 1≤n≤10. Your output shall look like:  The factorial of 1 is 1  The factorial of 2 is 2  ...  The factorial of 10 is 3628800  Modify your program (called **FactorialInt**), to list all the factorials, that can be expressed as an int (i.e., 32-bit signed integer). Your output shall look like:  The factorial of 1 is 1  The factorial of 2 is 2  ...  The factorial of 12 is 479001600  The factorial of 13 is out of range  Hints: Overflow occurs for Factorial(n+1) if (Integer.MAX\_VALUE / Factorial(n)) < (n+1).  Modify your program again (called **FactorialLong**) to list all the factorial that can be expressed as a long (64-bit signed integer). The maximum value for long is kept in a constant calledLong.MAX\_VALUE.  **Exercise NumberConversion:** Write a method call toRadix() which converts a positive integer from one radix into another. The method has the following header:  public static String toRadix(String in, int inRadix, int outRadix) // The input and output are treated as String.  Write a program called NumberConversion, which prompts the user for an input number, an input radix, and an output radix, and display the converted number. The output shall look like:  Enter a number and radix: A1B2  Enter the input radix: 16  Enter the output radix: 2  "A1B2" in radix 16 is "1010000110110010" in radix 2.  **Exercise NumberGuess:** Write a program called **NumberGuess** to play the number guessing game. The program shall generate a random number between 0 and 99. The player inputs his/her guess, and the program shall response with "Try higher", "Try lower" or "You got it in n trials" accordingly. For example:  > **java NumberGuess**  Key in your guess:  **50**  Try higher  **70**  Try lower  **65**  Try lower  "  You got it in 4 trials!  Hints: Use Math.random() to produce a random number in double between 0.0 and (less than) 1.0. To produce an int between 0 and 99, use:  int secretNumber = (int)(Math.random()\*100);  **Exercise WordGuess:** Write a program called WordGuess to guess a word by trying to guess the individual characters. The word to be guessed shall be provided using the command-line argument. Your program shall look like:  > **java WordGuess testing**  Key in one character or your guess word: **t**  Trail 1: t\_\_t\_\_\_  Key in one character or your guess word: **g**  Trail 2: t\_\_t\_\_g  Key in one character or your guess word: **e**  Trail 3: te\_t\_\_g  Key in one character or your guess word: **testing**  Trail 4: Congratulation!  You got in 4 trials  Hints:   * Set up a boolean array to indicate the positions of the word that have been guessed correctly. * Check the length of the input String to determine whether the player enters a single character or a guessed word. If the player enters a single character, check it against the word to be guessed, and update the boolean array that keeping the result so far. * Try retrieving the word to be guessed from a text file (or a dictionary) randomly.   **Exercise DateUtil:** Complete the following methods in a class called DateUtil:   * boolean isLeapYear(int year): returns true if the given year is a leap year. A year is a leap year if it is divisible by 4 but not by 100, or it is divisible by 400. * boolean isValidDate(int year, int month, int day): returns true if the given year, month and day constitute a given date. Assume that year is between 1 and 9999, month is between 1 (Jan) to 12 (Dec) and day shall be between 1 and 28|29|30|31 depending on the month and whether it is a leap year. * int getDayOfWeek(int year, int month, int day): returns the day of the week, where 0 for SUN, 1 for MON, ..., 6 for SAT, for the given date. Assume that the date is valid. * String toString(int year, int month, int day): prints the given date in the format "xxxday d mmm yyyy", e.g., "Tuesday 14 Feb 2012". Assume that the given date is valid.   To find the day of the week (Reference: Wiki "Determination of the day of the week"):   1. Based on the first two digit of the year, get the number from the following "century" table.  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | **1700-** | **1800-** | **1900-** | **2000-** | **2100-** | **2200-** | **2300-** | **2400-** | | 4 | 2 | 0 | 6 | 4 | 2 | 0 | 6 |  1. Take note that the entries 4, 2, 0, 6 repeat. 2. Add to the last two digit of the year. 3. Add to "the last two digit of the year divide by 4, truncate the fractional part". 4. Add to the number obtained from the following month table:  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | **Jan** | **Feb** | **Mar** | **Apr** | **May** | **Jun** | **Jul** | **Aug** | **Sep** | **Oct** | **Nov** | **Dec** | | Non-Leap Year | 0 | 3 | 3 | 6 | 1 | 4 | 6 | 2 | 5 | 0 | 3 | 5 | | Leap Year | 6 | 2 | same as above | | | | | | | | | |  1. Add to the day. 2. The sum modulus 7 gives the day of the week, where 0 for SUN, 1 for MON, ..., 6 for SAT.   For example: 2012, Feb, 17  (6 + 12 + 12/4 + 2 + 17) % 7 = 5 (Fri)  /\* Utilities for Date Manipulation \*/  public class **DateUtil** {    // Month's name – for printing  public static String strMonths[]  = {"Jan", "Feb", "Mar", "Apr", "May", "Jun",  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};    // Number of days in each month (for non-leap years)  public static int daysInMonths[]  = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};    // Returns true if the given year is a leap year  public static boolean isLeapYear(int year) { ...... }    // Return true if the given year, month, day is a valid date  // year: 1-9999  // month: 1(Jan)-12(Dec)  // day: 1-28|29|30|31. The last day depends on year and month  public static boolean isValidDate(int year, int month, int day) { ...... }    // Return the day of the week, 0:Sun, 1:Mon, ..., 6:Sat  public static int getDayOfWeek(int year, int month, int day) { ...... }    // Return String "xxxday d mmm yyyy" (e.g., Wednesday 29 Feb 2012)  public static String printDate(int year, int month, int day) { ...... }    public static void main(String[] args) {  System.out.println(isLeapYear(1900)); // false  System.out.println(isLeapYear(2000)); // true  System.out.println(isLeapYear(2011)); // false  System.out.println(isLeapYear(2012)); // true    System.out.println(isValidDate(2012, 2, 29)); // true  System.out.println(isValidDate(2011, 2, 29)); // false  System.out.println(isValidDate(2099, 12, 31)); // true  System.out.println(isValidDate(2099, 12, 32)); // true    System.out.println(getDayOfWeek(1982, 4, 24)); // 6:Sat  System.out.println(getDayOfWeek(2000, 1, 1)); // 6:Sat  System.out.println(getDayOfWeek(2054, 6, 19)); // 5:Fri  System.out.println(getDayOfWeek(2012, 2, 17)); // 5:Fri    System.out.println(toString(2012, 2, 14)); // Tuesday 14 Feb 2012  }  }  You can compare the day obtained with the Java's Calendar class as follows:  // Construct a Calendar instance with the given year, month and day  Calendar cal = new GregorianCalendar(year, month - 1, day); // month is 0-based  // Get the day of the week number: 1 (Sunday) to 7 (Saturday)  int dayNumber = cal.get(Calendar.DAY\_OF\_WEEK);  String[] calendarDays = { "Sunday", "Monday", "Tuesday", "Wednesday",  "Thursday", "Friday", "Saturday" };  // Print result  System.out.println("It is " + calendarDays[dayNumber - 1]);  The calendar we used today is known as Gregorian calendar, which came into effect in October 15, 1582 in some countries and later in other countries. It replaces the Julian calendar. 10 days were removed from the calendar, i.e., October 4, 1582 (Julian) was followed by October 15, 1582 (Gregorian). The only difference between the Gregorian and the Julian calendar is the "leap-year rule". In Julian calendar, every four years is a leap year. In Gregorian calendar, a leap year is a year that is divisible by 4 but not divisible by 100, or it is divisible by 400, i.e., the Gregorian calendar omits century years which are not divisible by 400. Furthermore, Julian calendar considers the first day of the year as march 25th, instead of January 1st.  This above algorithm work for Gregorian dates only. It is difficult to modify the above algorithm to handle pre-Gregorian dates. A better algorithm is to find the number of days from a known date. 8.  Exercises on Number Theory **Exercise (Perfect and Deficient Numbers):** A positive integer is called a perfect number if the sum of all its factors (excluding the number itself, i.e., proper divisor) is equal to its value. For example, the number 6 is perfect because its proper divisors are 1, 2, and 3, and 6=1+2+3; but the number 10 is not perfect because its proper divisors are 1, 2, and 5, and 10≠1+2+5.  A positive integer is called a deficient number if the sum of all its proper divisors is less than its value. For example, 10 is a deficient number because 1+2+5<10; while 12 is not because1+2+3+4+6>12.  Write a method called isPerfect(int posInt) that takes a positive integer, and return true if the number is perfect. Similarly, write a method called isDeficient(int posInt) to check for deficient numbers.  Using the methods, write a program called PerfectNumberList that prompts user for an upper bound (a positive integer), and lists all the perfect numbers less than or equal to this upper bound. It shall also list all the numbers that are neither deficient nor perfect. The output shall look like:  Enter the upper bound: **1000**  These numbers are perfect:  6 28 496  [3 perfect numbers found (0.30%)]    These numbers are neither deficient nor perfect:  12 18 20 24 30 36 40 42 48 54 56 60 66 70 72 78 80 ......  [246 numbers found (24.60%)]  **Exercise (Primes):** A positive integer is a prime if it is divisible by 1 and itself only. Write a method called isPrime(int posInt) that takes a positive integer and returns true if the number is a prime. Write a program called PrimeList that prompts the user for an upper bound (a positive integer), and lists all the primes less than or equal to it. Also display the percentage of prime (up to 2 decimal places). The output shall look like:  Please enter the upper bound: **10000**  1  2  3  ......  ......  9967  9973  [1230 primes found (12.30%)]  Hints: To check if a number n is a prime, the simplest way is try dividing n by 2 to √n.  **Exercise (Prime Factors):** Write a method isProductOfPrimeFactors(int posInt) that takes a positive integer, and return true if the product of all its prime factors (excluding 1 and the number itself) is equal to its value. For example, the method returns true for 30 (30=2×3×5) and false for 20 (20≠2×5). You may need to use the isPrime() method in the previous exercise.  Write a program called PerfectPrimeFactorList that prompts user for an upper bound. The program shall display all the numbers (less than or equal to the upper bound) that meets the above criteria. The output shall look like:  Enter the upper bound: **100**  These numbers are equal to the product of prime factors:  1 6 10 14 15 21 22 26 30 33 34 35 38 39 42 46 51 55 57 58 62 65 66 69 70 74 77 78 82 85 86 87 91 93 94 95  [36 numbers found (36.00%)]  **Exercise (Greatest Common Divisor):** One of the earlier known algorithms is the Euclid algorithm to find the GCD of two integers (developed by the Greek Mathematician Euclid around 300BC). By definition, GCD(a, b) is the greatest factor that divides both a and b. Assume that a and b are positive integers, and a≥b, the Euclid algorithm is based on these two properties:  GCD(a, 0) = a  GCD(a, b) = GCD(b, a mod b), where (a mod b) denotes the remainder of a divides by b.  For example,  GCD(15, 5) = GCD(5, 0) = 5  GCD(99,88) = GCD(88,11) = GCD(11,0) = 11  GCD(3456,1233) = GCD(1233,990) = GCD(990,243) = GCD(243,18) = GCD(18,9) = GCD(9,0) = 9  The pseudocode for the Euclid algorithm is as follows:  GCD(a, b) // assume that a ≥ b  while (b != 0) {  // Change the value of a and b: a ← b, b ← a mod b, and repeat until b is 0  temp ← b  b ← a mod b  a ← temp  }  // after the loop completes, i.e., b is 0, we have GCD(a, 0)  GCD is a  Write a method called gcd() with the following signature:  public static int gcd(int a, int b)  Your methods shall handle arbitrary values of a and b, and check for validity.  TRY: Write a recursive version called gcdRecursive() to find the GCD. | | |  | | --- | | **0. Serialization**    :    [**Link**](http://www.jusfortechies.com/java/core-java/serialization.php) | | | |  |

# Serialization in Java

Serialization is the process of converting an object's state (including its references) to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time. Simple......Coverting an object to bytes and bytes back to object. So when is serialization used? Serialization is used when you want to persist the object. It is also used by RMI to pass objects between JVMs, either as arguments in a method invocation from a client to a server or as return values from a method invocation. In general, serialization is used when we want the object to exist beyond the lifetime of the JVM.

*Lets see couple of different scenarios (examples) where we use serialization.*

* 1. Banking example: When the account holder tries to withdraw money from the server through ATM, the account holder information along with the withdrawl details will be serialized (marshalled/flattened to bytes) and sent to server where the details are deserialized (unmarshalled/rebuilt the bytes)and used to perform operations. This will reduce the network calls as we are serializing the whole object and sending to server and further request for information from client is not needed by the server.

* 2. Stock example: Lets say an user wants the stock updates immediately when he request for it. To achieve this, everytime we have an update, we can serialize it and save it in a file. When user requests the information, deserialize it from file and provide the information. This way we dont need to make the user wait for the information until we hit the database, perform computations and get the result.

Here are some uses of serialization

* To persist data for future use.
* To send data to a remote computer using such client/server Java technologies as RMI or socket programming.
* To "flatten" an object into array of bytes in memory.
* To exchange data between applets and servlets.
* To store user session in Web applications.
* To activate/passivate enterprise java beans.
* To send objects between the servers in a cluster.

So far we saw what and when serialization is used.

*Lets see now how serialization is performed in java.*

Java provides Serialization API, a standard mechanism to handle object serialization. To persist an object in java, the first step is to flatten the object. For that the respective class should implement "java.io.Serializable" interface. Thats it. We dont need to implement any methods as this interface do not have any methods. This is a marker interface/tag interface. Marking a class as Serializable indicates the underlying API that this object can be flattened.

public class SerialClass implements Serializable {

private Date currentTime;

public SerialClass() {

currentTime = Calendar.getInstance().getTime();

}

public Date getCurrentTime() {

return currentTime;

}

}

Now you marked the object for flattening. Next step is to actually persist the object. To persist an object you need to use node stream to write to file systems or transfer a flattened object across a network wire and have it be rebuilt on the other side. You can use java.io.ObjectOutputStream class, a filter stream which is a wrapper around a lower-level byte stream.

So to write an object you use "writeObject(<<instance>>)" method of "java.io.ObjectOutputStream" class and to read an object you use "readObject()" method of "java.io.ObjectOutputStream" class. "readObject()" can read only serialized object, that means if the class does not implement "java.io.Serializable" interface, "readObject()" cannot read that object.

//Class to persist the time in a flat file time.ser

public class PersistSerialClass {

public static void main(String [] args) {

String filename = "time.ser";

if(args.length > 0){

filename = args[0];

}

PersistSerialClass time = new PersistSerialClass();

FileOutputStream fos = null;

ObjectOutputStream out = null;

try{

fos = new FileOutputStream(filename);

out = new ObjectOutputStream(fos);

out.writeObject(time);

out.close();

}catch(IOException ex){

ex.printStackTrace();

}

}

}

//Class to read the time from a flat file time.ser

public class ReadSerialClass {

public static void main(String [] args) {

String filename = "time.ser";

if(args.length > 0){

filename = args[0];

}

PersistSerialClass time = null;

FileInputStream fis = null;

ObjectInputStream in = null;

try{

fis = new FileInputStream(filename);

in = new ObjectInputStream(fis);

time = (PersistSerialClass)in.readObject();

in.close();

}catch(IOException ex){

ex.printStackTrace();

}catch(ClassNotFoundException cnfe){

cnfe.printStackTrace();

}

// print out restored time

System.out.println("Restored time: " + time.getTime());

// print out the current time

System.out.println("Current time: "

+ Calendar.getInstance().getTime());

}

}

When you serialize an object, only the object's state will be saved, not the object's class file or methods.

When you serialize the above example class, the serialized class will look like below. Surprising.. isn't it? Yes, when you serialized a 2 byte object, you see 51 bytes serialized file. How did it convert to 51 bytes file? To know this,

*Let's see step by step on how the object is serialized and de-serialized.*

So when an object is serailized

* First it writes out the serialization stream magic data - What is serialization stream magic data? This is nothing but the STREAM\_MAGIC and STREAM\_VERSION data (static data) so that JVM can deserialize it when it has to. The STRAM\_MAGIC will be "AC ED" and the STREAM\_VERSION will be the version of the JVM.
* Then it writes out the metadata (description) of the class associated with an instance. So in the below example after writing out the magic data, it writes out the description of "SerialClass" class. What does this description include? It includes the length of the class, the name of the class, serialVersionUID (or serial version), various flags and the number of fields in this class.
* Then it recursively writes out the metadata of the superclass until it finds java.lang.object. Again in the below example, it writes out the description of "SerialParent" and "SerialParentParent" classes.
* Once it finishes writing the metadata information, it then starts with the actual data associated with the instance. But this time, it starts from the top most superclass. So it starts from "SerialParentParent", then writes out "SerialParent".
* Finally it writes the data of objects associated with the instance starting from metadata to actual content. So here it starts writing the metadata for the class Contain and then the contents of it as usual recursively.

class SerialParentParent implements Serializable {

int parentParentVersion = 10;

}

class SerialParent implements Serializable {

int parentVersion = 11;

}

class Contain implements Serializable{

int containVersion = 20;

}

public class SerialClass extends SerialParent implements Serializable {

int version = 12;

Contain con = new Contain();

public int getVersion() {

return version;

}

public static void main(String args[]) throws IOException {

FileOutputStream fos = new FileOutputStream("temp.ser");

ObjectOutputStream oos = new ObjectOutputStream(fos);

SerialClass st = new SerialClass();

oos.writeObject(st);

oos.flush();

oos.close();

}

}

*How to customize the default protocol?*

MMmmmm.. Now its getting more interesting. Lets say, you need to perform some specific operations in the constructor when you are instantiating the class but you cant perform those operations when you deserialize the object because constructor wont be called when an object is de-serialized. Here we are restoring an object but not reconstructing an object. Then how will you call or perform those operations when you desrialize the object? Well, you have a way here and its simple too. You can enhance the normal process by providing two methods inside your serializable class. Those methods are:  
  
*private void writeObject(ObjectOutputStream out) throws IOException;*  
  
*private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException;*  
  
Notice that both methods are declared private and ofcourse they must be declared private, proving that neither method is inherited and overridden or overloaded. The trick here is that the virtual machine will automatically check to see if either method is declared during the corresponding method call. The virtual machine can call private methods of your class whenever it wants but no other objects can. Thus, the integrity of the class is maintained and the serialization protocol can continue to work as normal.

public class SerialClass implements Serializable {

private Date currentTime;

public SerialClass() {

calculateCurrentTime();

}

public Date getCurrentTime() {

return currentTime;

}

private Date calculateCurrentTime(){

currentTime = Calendar.getInstance().getTime();

}

private void writeObject(ObjectOutputStream out) throws IOException {

out.defaultWriteObject();

}

private void readObject(ObjectInputStream in)

throws IOException, ClassNotFoundException{

// our "pseudo-constructor"

in.defaultReadObject();

// now perfrom same operation you need to do in constructor

calculateCurrentTime();

}

}

Ooops. I mentioned earlier that for a class to be serializable either the class should implement "Serializable" interface or one of its super class should implement "Serializable" interface. Now what if you dont want to serialize one of the sub class of a serializable class? You have a way here tooo. To stop the automatic serialization, you can once again use the private methods to just throw the NotSerializableException in your class.

private void writeObject(ObjectOutputStream out) throws IOException{

throw new NotSerializableException("Dont Serialize");

}

private void readObject(ObjectInputStream in) throws IOException{

throw new NotSerializableException("Dont Serialize");

}

*Well... One more way to serialize the object - the Externalizable Interface*

Again there is one more way to serialize the object - create your own protocol with the Externalizable interface. Instead of implementing the Serializable interface, you can implement Externalizable, which contains two methods:   
  
*public void writeExternal(ObjectOutput out) throws IOException;*   
  
*public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;*   
  
The Externalization is discussed as separate topic. Check it out [here](http://www.jusfortechies.com/java/core-java/externalization.php)or check the menu.

*How not to serialize some fields in a serializble object?*

Sometimes you dont want to serialize/store all the fields in the object. Say some fields you want to hide to preserve the privacy or some fields you may want to read only from master data, then you dont seriaalize them. To do this, you just need to declare a field as ***transient*** field.

transient private int checkPoint;

Also the static fields are not serialized. Actually there is no point in serializing static fields as static fields do not represent object state but represent class state and it can be modified by any other object. Lets assume that you have serialized a static field and its value and before deserialization of the object, the static field value is changed by some other object. Now the static field value that is serialized/stored is no more valid. Hence it make no point in serializing the static field.

Apart from declaring the field as transient, there is another tricky way of controlling what fields can be serialized and what fields cannot be. This is by overriding the writeObject() method while serialization and inside this method you are responsible for writing out the appropriate fields. When you do this, you may have to override readObject() method as well. This sound similar to using Externalizable where you will write writeExternal() and readExternal() methods but anyways lets not take this route as this is not a neat route.

Note that serialization does not care about access modifiers. It serializes all private, public and protected fields.

*Nonserializable objects*

Earlier we discussed about not serializing certain fields in a serializable object and why it may be needed sometimes. But now lets see why certain objects should not be serialized? As you know, the Object class does not implement Serializable interface and hence any object by default is not serializable. To make an object serializable, the respective class should explicitly implement Serializable interface. However certain system classes defined by java like "Thread", "OutputStream", "Socket" are not serializable. Why so? Lets take a step back - now what is the use of serializing the Thread running in System1 JVM using System1 memory and then deserializing it in System2 and trying to run in System2 JVM. Makes no sense right! Hence these classes are not serializable.

Ok. So far so good. Now what if you want to serialize an object containing an instance of Thread? Simple. Declare the Thread instance as transient.

public class SerialClass implements Serializable, Runnable {

transient private Thread myThread;

public PersistentAnimation(int animationSpeed) {

...

...

}

}

*Versioning issues*

One very important item to look at is the versioning issue. Sometimes you wil get "java.io.InvalidClassException" but when you check the class (it will be Serializable class), you will find nothing wrong with it. Then what is causing this exception to be thrown? Ok. Here it is. You create a Serializable class, instantiate it, and write it out to an object stream. That flattened object sits in the file system for some time. Meanwhile, you update the class file by adding a new field. Then try to read the flattened object. InvalidClassException is thrown because all persistent-capable classes are automatically given a unique identifier. If the identifier of the class does not equal the identifier of the flattened object, the exception will be thrown and when you update the class with a new field, a new identifier will be generated.

To fix this issue, manually add the identifier to the class. The identifier that is part of all classes is maintained in a field called serialVersionUID. If you wish to control versioning, you simply have to provide the serialVersionUID field manually and ensure it is always the same, no matter what changes you make to the classfile. More about it is discussed in separate topic. [Check it here.](http://www.jusfortechies.com/java/core-java/serialVersionUID.php)

*Performance Issues/Improvement with Serialization*

The default way of implementing the serialization (by implementing the Serializable interface) has performance glitches. Say you have to write an object 10000 times in a flat file through serialization, this will take much more (alomost double) the time than it takes to write the same object 10000 times to console. To overcome this issue, its always better to write your custom protocol instead of going for default option.

Also always note to close the streams (object output and input streams). The object references are cached in the output stream and if the stream is not closed, the system may not garbage collect the objects written to a stream and this will result in poor performance.

Using Serialization always have performance issues? Nope... Let me give you a situation where it is used for performance improvement. Lets assume you have an application that should display a map and pointing to different areas in the map should highlight those areas in different color. Since all these are images, when you point to each location, loading an image each time will slow the application heavily. To resolve this issue, serialization can be used. So here since the images wont change, you can serialize the image object and the respective points on the map (x and y co-ordinates) and deserialize it as and when necessary. This improves the performance greatly.

*What happens to inner classes? We forgot all about it.*

Yes, you can serialize inner classes by implementing the Serializable interface but it has some problems. Inner classes (declared in a non-static context) will always contain implicit references to their enclosing classes and these references are always non-transient. So, while object serialization process of inner classes, the enclosing classes will also be serialized. Now the problem is that the synthetic fields generated by Java compilers to implement inner classes are pretty much implementation dependent and hence we may face compatibility issues while deserialization on a different platform having a .class file generated by a different Java compiler. The default serialVerionUID may also be different in such cases. Not only this, the names assigned to the local and anonymous inner classes are also implementation dependent. Thus, we see that object serialization of inner classes may pose some unavoidable compatibility issues and hence the serialization of inner classes is strongly discouraged.

**Chapter 13:Object Equality**

In this tutorial we will learn the difference between String,StringBuilder and StringBuffer. Also we will learn Object as a superclass.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | |  | | --- | | **0. Object: The Cosmic Superclass**    :    [**Link**](http://docs.oracle.com/javase/tutorial/java/IandI/objectclass.html) |  Object as a Superclass The [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) class, in the java.lang package, sits at the top of the class hierarchy tree. Every class is a descendant, direct or indirect, of the Object class. Every class you use or write inherits the instance methods of Object. You need not use any of these methods, but, if you choose to do so, you may need to override them with code that is specific to your class. The methods inherited from Object that are discussed in this section are:   * protected Object clone() throws CloneNotSupportedException       Creates and returns a copy of this object. * public boolean equals(Object obj)       Indicates whether some other object is "equal to" this one. * protected void finalize() throws Throwable       Called by the garbage collector on an object when garbage       collection determines that there are no more references to the object * public final Class getClass()       Returns the runtime class of an object. * public int hashCode()       Returns a hash code value for the object. * public String toString()       Returns a string representation of the object.   The notify, notifyAll, and wait methods of Object all play a part in synchronizing the activities of independently running threads in a program, which is discussed in a later lesson and won't be covered here. There are five of these methods:   * public final void notify() * public final void notifyAll() * public final void wait() * public final void wait(long timeout) * public final void wait(long timeout, int nanos)   **Note:** There are some subtle aspects to a number of these methods, especially the clone method. The clone() Method If a class, or one of its superclasses, implements the Cloneable interface, you can use the clone() method to create a copy from an existing object. To create a clone, you write:  *aCloneableObject*.clone();  Object's implementation of this method checks to see whether the object on which clone() was invoked implements the Cloneable interface. If the object does not, the method throws a CloneNotSupportedException exception. Exception handling will be covered in a later lesson. For the moment, you need to know that clone() must be declared as  protected Object clone() throws CloneNotSupportedException  or:  public Object clone() throws CloneNotSupportedException  if you are going to write a clone() method to override the one in Object.  If the object on which clone() was invoked does implement the Cloneable interface, Object's implementation of the clone() method creates an object of the same class as the original object and initializes the new object's member variables to have the same values as the original object's corresponding member variables.  The simplest way to make your class cloneable is to add implements Cloneable to your class's declaration. then your objects can invoke the clone() method.  For some classes, the default behavior of Object's clone() method works just fine. If, however, an object contains a reference to an external object, say ObjExternal, you may need to override clone() to get correct behavior. Otherwise, a change in ObjExternal made by one object will be visible in its clone also. This means that the original object and its clone are not independent—to decouple them, you must override clone() so that it clones the object *and* ObjExternal. Then the original object referencesObjExternal and the clone references a clone of ObjExternal, so that the object and its clone are truly independent. The equals() Method The equals() method compares two objects for equality and returns true if they are equal. The equals() method provided in the Object class uses the identity operator (==) to determine whether two objects are equal. For primitive data types, this gives the correct result. For objects, however, it does not. The equals() method provided byObject tests whether the object *references* are equal—that is, if the objects compared are the exact same object.  To test whether two objects are equal in the sense of *equivalency* (containing the same information), you must override the equals() method. Here is an example of a Bookclass that overrides equals():  public class Book {  ...  public boolean equals(Object obj) {  if (obj instanceof Book)  return ISBN.equals((Book)obj.getISBN());  else  return false;  }  }  Consider this code that tests two instances of the Book class for equality:  // Swing Tutorial, 2nd edition  Book firstBook = new Book("0201914670");  Book secondBook = new Book("0201914670");  if (firstBook.equals(secondBook)) {  System.out.println("objects are equal");  } else {  System.out.println("objects are not equal");  }  This program displays objects are equal even though firstBook and secondBook reference two distinct objects. They are considered equal because the objects compared contain the same ISBN number.  You should always override the equals() method if the identity operator is not appropriate for your class.  **Note:** If you override equals(), you must override hashCode() as well. The finalize() Method The Object class provides a callback method, finalize(), that *may be* invoked on an object when it becomes garbage. Object's implementation of finalize() does nothing—you can override finalize() to do cleanup, such as freeing resources.  The finalize() method may be called automatically by the system, but when it is called, or even if it is called, is uncertain. Therefore, you should not rely on this method to do your cleanup for you. For example, if you don't close file descriptors in your code after performing I/O and you expect finalize() to close them for you, you may run out of file descriptors. The getClass() Method You cannot override getClass.  The getClass() method returns a Class object, which has methods you can use to get information about the class, such as its name (getSimpleName()), its superclass (getSuperclass()), and the interfaces it implements (getInterfaces()). For example, the following method gets and displays the class name of an object:  void printClassName(Object obj) {  System.out.println("The object's" + " class is " +  obj.getClass().getSimpleName());  }  The [Class](https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html) class, in the java.lang package, has a large number of methods (more than 50). For example, you can test to see if the class is an annotation (isAnnotation()), an interface (isInterface()), or an enumeration (isEnum()). You can see what the object's fields are (getFields()) or what its methods are (getMethods()), and so on. The hashCode() Method The value returned by hashCode() is the object's hash code, which is the object's memory address in hexadecimal.  By definition, if two objects are equal, their hash code *must also* be equal. If you override the equals() method, you change the way two objects are equated and Object's implementation of hashCode() is no longer valid. Therefore, if you override the equals() method, you must also override the hashCode() method as well. The toString() Method You should always consider overriding the toString() method in your classes.  The Object's toString() method returns a String representation of the object, which is very useful for debugging. The String representation for an object depends entirely on the object, which is why you need to override toString() in your classes.  You can use toString() along with System.out.println() to display a text representation of an object, such as an instance of Book:  System.out.println(firstBook.toString());  which would, for a properly overridden toString() method, print something useful, like this:  ISBN: 0201914670; The Swing Tutorial; A Guide to Constructing GUIs, 2nd Edition | | |  | | --- | | **0. The toString() Method**    :    [**Link**](http://www.tutorialspoint.com/java/number_tostring.htm)  [**Link**](http://home.cogeco.ca/~ve3ll/jatutor7.htm#pr) |  Description: The method is used to get a String object representing the value of the Number Object.  If the method takes a primitive data type as an argument, then the String object representing the primitive data type value is return.  If the method takes two arguments, then a String representation of the first argument in the radix specified by the second argument will be returned. Syntax: All the variant of this method are given below:  String toString()  static String toString(int i) Parameters: Here is the detail of parameters:   * **i** -- An int for which string representation would be returned.  Return Value:  * **toString():** This returns a String object representing the value of **this** Integer. * **toString(int i):** This returns a String object representing the specified integer.  Example: public class Test{  public static void main(String args[]){  Integer x = 5;  System.out.println(x.toString());  System.out.println(Integer.toString(12));  }  }  This produces the following result:  5  12  The ***Java collections framework*** applies a unifying philosophy that adds operational functionality and dynamic growth to data structure object classes. Unification is accomplished through ***interfaces*** that each collection object ***inherits***. Various types of objects may be handled in a similar manner within collection framework classes. Functionality such as searches, sorts, insertion and deletion use highly efficient algorithms.  ***Exception handling*** is a method of ***trapping*** or coping with anticipated errors (system, data entry or calculation) and ***handling*** or dealing with them in a graceful manner. The ***Exception*** class of objects offers a rich group of subclasses to trap specific errors and recover from them.  ***Standard IO*** (input/output) operations ***stream*** data from keyboard, files, programs or devices into the program and from there to the screen, printer, files, programs or other devices.   |  |  |  | | --- | --- | --- | | * [**Collections & Mappings**](http://home.cogeco.ca/~ve3ll/jatutor7.htm#co) * [**Utilities and Algorithms**](http://home.cogeco.ca/~ve3ll/jatutor7.htm#ut) * [**Generic Type Parameters**](http://home.cogeco.ca/~ve3ll/jatutor7.htm#pa) | * [**Project: Arrays to Lists**](http://home.cogeco.ca/~ve3ll/jatutor7.htm#con) * [**Project: Concordance**](http://home.cogeco.ca/~ve3ll/jatutor7.htm#pr1) | * [**Exception Handling**](http://home.cogeco.ca/~ve3ll/jatutor7.htm#ex) * [**Effective Hierarchy**](http://home.cogeco.ca/~ve3ll/jatutor7.htm#hi) * [**Standard IO Streams**](http://home.cogeco.ca/~ve3ll/jatutor7.htm#io) |   **Collections and Mappings**  The ***collection interfaces*** are dynamic data structures similar to arrays. The interfaces are ***Collection, List, Set, SortedSet and Queue***. ***Collection*** is at the top of this hierarchy and includes the core methods ***add(obj), addAll(coll), clear(), contains(obj), containsAll(coll), equals(obj), hashCode(), isEmpty(), iterator(), remove(obj), removeAll(coll), retainAll(coll), size(), and toArray()***. ***List*** extends Collection for classes that are a sequence of elements (similar to an array). It adds the methods ***get(idx), indexOf(), lastIndexOf(), set(idx) and subList(i1,i2)*** and overrides appropriate methods. ***Set*** extends Collection by overriding the add() method to prevent duplicate objects. ***SortedSet*** extends Set to keep all contained objects in a ordered manner. Methods include ***comparator(), first(), headSet(obj), last(), subSet(o1,o2) and tailSet(obj)***. ***Queue*** puts objects in a line pending processing.  Concrete ***collection classes*** implement the appropriate interfaces and incorporate optimized data structures. The collection classes include ***ArrayList, HashSet, LinkedList and TreeSet***. ***ArrayList*** implements List and is similar to arrays but is ***dynamic*** (ie. its size does not have to be declared at compile time). Additional methods in ArrayList include ***lastIndexOf(obj)*** to return the object's position, ***get(index)*** to return an object at a specific position, and ***toArray()*** to convert a collection back to a regular array. ***HashSet*** implements the Set interface and uses a hash table for efficient storage. ***LinkedList*** implements the appropriate data structure which can be traversed and nodes inserted and deleted. ***TreeSet*** implements the SortedSet interface using a binary tree for storage, allowing rapid access to list elements.  **Note:** Collection classes are contained in the ***java.util*** package.  An example of how specific list objects can be located for access by indexing is:  // Staff is a list object  for (int x=0; x<emp.size(); x++) // by index number  {if (emp.get(x) instanceof Staff) // emp can be of several types  {Staff s=(Staff) emp.get(x);} } // cast it to new Staff object  An example of walking through a collection with an iterator is:  Iterator it=c.iterator(); // set up iterator for the collection  while (it.hasNext()) {this.add((String)it.next());} // check, then access  The ***mapping interfaces*** are Map and SortedMap. These are similar to 'associative arrays' in other languages.***Map*** defines a table that is a one-to-one relationship between a key and an element. No duplicate keys are allowed. ***SortedMap*** maintains order within the mapping.  ***HashMap*** and ***TreeMap*** are concrete ***mapping*** classes. TreeMap orders the mapping  key while HashMap does not.  **Note:** A ***ClassCastException*** error is generated at run-time if objects are incompatible such as adding an incompatible object to a collection.  **Note:** All collection objects are ***unsynchronized***. If [**threaded**](http://home.cogeco.ca/~ve3ll/jatutore.htm) programming is used, place the collection in a synchronization wrapper.  **Utilities and Algorithms**  The ***utility interfaces*** include Comparator, Iterator and ListIterator. Their methods can be customized by overriding. ***Comparator*** defines how two objects are compared. It has two methods, ***compare(obj1,obj2)*** and***equals(obj)*** ***Iterator*** and ***ListIterator*** enumerate objects within a list (similar to an array index). Iterator contains the methods ***next(), hasNext()*** and ***remove()***. ListIterator has the additional methods of ***hasPrevious(), previous()*** and ***set()***.  Several ***algorithms*** are defined as ***static methods*** within the collections interface. They are ***binarySearch(), binarySort(), copy(), enumeration(), fill(), max(), min(), nCopies(), reverse(), reverseOrder(), shuffle(), singleton(), sort(), sychronizedCollection(), synchronizedList(), synchronizedMap(), synchronizedSet(), synchronizedSortedMap(), synchronizedSortedSet(), unmodifiableCollection(), unmodifiablelist(), unmodifiableMap(), unmodifiableset(), unmodifiableSortedMap()*** and ***unmodifiableSortedSet()***.  **Generic Type Parameters**  [**Generics**](https://en.wikipedia.org/wiki/Generics_in_Java) allow a type or method to operate on objects of various types while providing compile-time type safety. Objects can be ***type checked*** at compile time by using ***parameterized types*** to avoid casting errors at runtime.  LinkedList<Integer> myIntList=new LinkedList<Integer>();  myIntList.add(new Integer(0));  Integer x=myIntList.iterator().next();  The type declaration inside the angle bracket ensures checking at compile time. No casting is required! Types specified must be class types. Primitives must use their wrapper objects. Mappings such as HashMap and TreeMap specify objects by using an ordered list (eg. ***<String,Float>***).  **Warning:** Use wildcard typing ***<?>*** at your own peril!!!  **Project: Convert Arrays to Lists**  Static arrays are easily converted to ***dynamic lists***. This can be appreciated when [**WordCount2**](http://home.cogeco.ca/~ve3ll/jatutor8.htm#sc) is converted into [**wcPlus**](http://home.cogeco.ca/~ve3ll/jatutor8.htm#sc) allowing an unlimited number of unique words instead of a precompiled limit. A single list of word objects is used instead of parallel word and count arrays. Iterators are used instead of indexes when walking through the list.  **Project: Concordance**  A [**concordance**](https://en.wikipedia.org/wiki/Concordance) produces an index of words and line numbers of occurrence within a document. This project uses the [**StringTokenizer**](http://home.cogeco.ca/~ve3ll/jatutor6.htm#to) class to provide the words/locations to be indexed and ***Java collections*** classes to add dynamic size ***(ArrayList)*** and sorting ***(TreeMap)***. The report should be in columns for readability. Both[**concordance.java**](http://home.cogeco.ca/~ve3ll/jatutor8.htm#sc) (command line) and [**concord.java**](http://home.cogeco.ca/~ve3ll/jatutor8.htm#sc) (GUI based) use [**file IO**](http://home.cogeco.ca/~ve3ll/jatutor8.htm) techniques. ***concord.java*** is also used as an example of how an application is deployed using [**Web Start**](http://home.cogeco.ca/~ve3ll/jatutori.htm#web).  **Exception Handling**  ***Exceptions*** are objects that describe any error caused by an ***external*** resource not being available or an***internal*** processing problem. They are passed to exception handlers written by the programmer to enable graceful recovery. If the handler has not been written, the program terminates with a display of the Exception class. There are many exception classes such as ***IOException*** and ***NumberFormatException***.  **Note:** Exception handling classes are contained in the ***java.lang*** package.  Java uses the ***try - catch - finally*** syntax to test (ie. try) a section of code and if an ***exception*** error occurs in that region, to trap (ie. catch) the error. Any number of catches can be set up for various ***exception types***. The***finally*** keyword can be used to provide a block of code that is performed regardless of whether an exception is signaled or not. The syntax is:  try  {  // tested statement(s);  }  catch(ExceptionName e1)  {  // trap handler statement(s);  }  catch(ExceptionName e2) // any number of catch statements  {  // display exception to screen  System.out.println("Exception: "+e2);  }  finally  {  // always executed block  }  The ***throws clause*** is used to pass a possible exception up the stack (processing chain). The Java compiler is aware of how some methods may cause specific exceptions and it forces one to deal with these immediately. If you choose ***not*** to write an exception handler then use the ***throws java.io.xxxException*** clause on the surrounding ***method*** to abdicate responsibility. For example ***System.in.read()*** gives a compile error for IOException. Add the throws clause to the surrounding method to pass the error up to the next level (or else write your own catch/try handler).  **Note:** ***xxxException properties*** are found in the java.io library.  The ***throw keyword*** (note the singular form) is used to force an exception. It can also pass a custom message to the exception handling module. For example:  throw new java.io.FileNotFoundException("Not found: beatles.txt");  **Effective Exception Class Hierarchy**  The exception classes are arranged in a hierarchy. Handlers (or catches) for specific exceptions should always be written prior to the generic handler. For example since ***FileNotFoundException*** is a child of ***IOException***, a catch for FileNotFoundException should occur before the one for IOException. The latter handler catches those exceptions that are missed by individual child handlers. And a generic handler for ***Exception*** would cover any missing situation.  FileInputStream fis=null; // declare in outer block  try  {  fis=new FileInputStream(new File (args[0])); // uses cmd line  int ch;  while (ch=fis.read() != -1)  {System.out.print((char) ch);}  fis.close();  System.out.println("");  }  catch (FileNotFoundException e)  {  System.out.println("File not found!");  try  {  fis.close();  }  catch(IOException ioe){} // disregard close failure  }  catch (IOException e)  {  try  {  fis.close();  }  catch(IOException ioe){} // disregard close failure  System.out.println("Unable to read file!");  }  **Standard IO Streams**  Modern program design uses a ***stream*** metaphor for information flow. Streams of data flow from sources (inputs) through applications (programs) into sinks (outputs). Storage reservoirs (files) are used when needed. The ***standard*** input and output devices are the user console by default but can be ***redirected*** by the ***operating system*** to files, printers or other devices using the symbols <, > and |. [**Filters**](https://en.wikipedia.org/wiki/Filter_%28Unix%29) are applications that assume standard io streams.  ***System.in*** and ***System.out*** objects read and write from the standard stream. ***System.err*** always refers to the console (no redirection allowed). The method ***read()*** returns an ***integer*** that is ASCII of the input character. It can also throw an ***IOException*** [**exception**](http://home.cogeco.ca/~ve3ll/jatutor7.htm#ex) error. The output methods are ***print(string)***, ***println(string)*** and***printf(format, object\_list)***[**[using c-like syntax formatting]**](https://docs.oracle.com/javase/1.5.0/docs/api/java/util/Formatter.html).  public void main (String args[]) throws java.io.IOException  { int ch;  System.out.println("Enter text: ");  while (ch=System.in.read()!='\n') {  System.out.println(ch); // shows it reads ASCII  System.out.println((char)ch); }  There are alternate ways for user interaction. Refer to [**scanner class**](http://home.cogeco.ca/~ve3ll/jatutor8.htm#to) for stdio streams, [**file io**](http://home.cogeco.ca/~ve3ll/jatutor8.htm) for basic file management or [**file choosers**](http://home.cogeco.ca/~ve3ll/jatutord.htm#fc) for visual file interfaces using Swing objects.   |  |  | | --- | --- | | * [**Tutorials - Table of Contents**](http://home.cogeco.ca/~ve3ll/jatutor0.htm) * [**Tutorial 1 - Structure / Syntax**](http://home.cogeco.ca/~ve3ll/jatutor1.htm) * [**Tutorial 2 - Control Flow**](http://home.cogeco.ca/~ve3ll/jatutor2.htm) * [**Tutorial 3 - Encapsulation & Classes**](http://home.cogeco.ca/~ve3ll/jatutor3.htm) * [**Tutorial 4 - Inheritance & Polymorphism**](http://home.cogeco.ca/~ve3ll/jatutor4.htm) * [**Tutorial 5 - Java Class Libraries**](http://home.cogeco.ca/~ve3ll/jatutor5.htm) * [**Tutorial 6 - String Manipulation**](http://home.cogeco.ca/~ve3ll/jatutor6.htm) * [**Tutorial 7 - Collections, Exceptions, IO**](http://home.cogeco.ca/~ve3ll/jatutor7.htm) * [**Tutorial 8 - File IO**](http://home.cogeco.ca/~ve3ll/jatutor8.htm) * [**Tutorial 9 - GUI Swing Widgets**](http://home.cogeco.ca/~ve3ll/jatutor9.htm) | * [**Tutorial 10 - Intermediate Swing**](http://home.cogeco.ca/~ve3ll/jatutora.htm) * [**Tutorial 11 - Advanced Swing**](http://home.cogeco.ca/~ve3ll/jatutorb.htm) * [**Tutorial 12 - UI Design/Implementation**](http://home.cogeco.ca/~ve3ll/jatutorc.htm) * [**Tutorial 13 - GUI Views and Studies**](http://home.cogeco.ca/~ve3ll/jatutord.htm) * [**Tutorial 14 - Threads & Serialization**](http://home.cogeco.ca/~ve3ll/jatutore.htm) * [**Tutorial 15 - Java Networking**](http://home.cogeco.ca/~ve3ll/jatutorf.htm) * [**Tutorial 16 - Graphics and Imaging**](http://home.cogeco.ca/~ve3ll/jatutorg.htm) * [**Tutorial 17 - Testing Applications**](http://home.cogeco.ca/~ve3ll/jatutorh.htm) * [**Tutorial 18 - Apps Deployment**](http://home.cogeco.ca/~ve3ll/jatutori.htm) * [**Appendices**](http://home.cogeco.ca/~ve3ll/jatutorx.htm) |   [**JR's HomePage**](http://home.cogeco.ca/~ve3ll/home.htm) | | |  | | --- | | **0. equals()**    :    [**Link**](http://www.tutorialspoint.com/java/java_string_equals.htm)  [**Link**](http://etutorials.org/Misc/programmers+guide+java+certification/Chapter+11.+Collections+and+Maps/Programming+Exercises/) |  Description: This method compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object. Syntax: Here is the syntax of this method:  public boolean equals(Object anObject) Parameters: Here is the detail of parameters:   * **anObject** -- the object to compare this String against.  Return Value :  * This method returns true if the String are equal; false otherwise.  Example: public class Test {  public static void main(String args[]) {  String Str1 = new String("This is really not immutable!!");  String Str2 = Str1;  String Str3 = new String("This is really not immutable!!");  boolean retVal;  retVal = Str1.equals( Str2 );  System.out.println("Returned Value = " + retVal );  retVal = Str1.equals( Str3 );  System.out.println("Returned Value = " + retVal );  }  }  This produces the following result:  Returned Value = true  Returned Value = true | | |  | | --- | | **0. String Pool**    :    [**Link**](http://www.dreamincode.net/forums/topic/187631-the-string-pool/)  [**Link**](http://www.programr.com/practice/programr211_promo_228/node/609) |   The String Pool Sun made an optimization that is rather confusing to many new Java programmers called the String Pool. It allows for Strings, which are one of the most used Objects to optimize themselves and save space. An important point to make is that the String Pool only applies to String literals, meaning their value was *assigned*, not constructed using a String constructor. Let me start off with an example:   public class Main {    public static void main(String[] args) {  String s1 = "abc";  String s2 = "abc";    System.out.println("s1 = " + s1);  System.out.println("s2 = " + s2);  System.out.println("s1 == s2? " + (s1 == s2));  System.out.println("s1.equals(s2)? " + (s1.equals  (s2)));  }    }  If you run this program, you get this:  **Quote**  s1 = abc s2 = abc s1 == s2? true s1.equals(s2)? true  Most beginners are told to only compare strings using the .equals() method, simply because it is safer for beginners, however, what is confusing about my program above is that the == operator actually said those two were the same memory location too. How is s1 == s2? I clearly defined 2 different variables! An important concept about Java that you may not know is that symbols (variable names) aren't actually the object that they are defined as. They actually hold a reference to the spot in memory where the actual object is kept. Since Strings are SO commonly used, Strings literals that are the same are given the same address so that it saves memory and doesn't have to make another. Think about it as if there can only be one of each String in there and anything matching is assigned a reference to that String. However, once all references are gone, then the object is erased. However, once they change, the addresses are different:  public class Main {    public static void main(String[] args) {  String s1 = "abc";  String s2 = "abc";    System.out.println("s1 = " + s1);  System.out.println("s2 = " + s2);  System.out.println("s1 == s2? " + (s1 == s2));  System.out.println("s1.equals(s2)? " + (s1.equals  (s2)));    s2 += "abc";    System.out.println("s1 = " + s1);  System.out.println("s2 = " + s2);  System.out.println("s1 == s2? " + (s1 == s2));  System.out.println("s1.equals(s2)? " + (s1.equals  (s2)));    }    }  This outputs this:  **Quote**  s1 = abc s2 = abc s1 == s2? true s1.equals(s2)? true s1 = abc s2 = abcabc s1 == s2? false s1.equals(s2)? false  HOWEVER, if for any reason you want the variables to not occupy the same location in memory, there are two ways to do this. First, you can use the String constructor, since those are not put into the pool:  public class Main {    public static void main(String[] args) {  String s1 = "abc";  String s2 = new String("abc");    System.out.println("s1 = " + s1);  System.out.println("s2 = " + s2);  System.out.println("s1 == s2? " + (s1 == s2));  System.out.println("s1.equals(s2)? " + (s1.equals  (s2)));    }    }  This code says that == is false. And just to prove a point, let's make both of them use constructors:  public class Main {    public static void main(String[] args) {  String s1 = new String("abc");  String s2 = new String("abc");    System.out.println("s1 = " + s1);  System.out.println("s2 = " + s2);  System.out.println("s1 == s2? " + (s1 == s2));  System.out.println("s1.equals(s2)? " + (s1.equals  (s2)));    }    }  That code yields:  **Quote**  s1 = abc s2 = abc s1 == s2? false s1.equals(s2)? true  OR you can assign them in different steps, forcing it to occupy a diffent address for each addition. (First they are the same, then s1 is moved to another spot to add on the "c" and the same for s2):  public class Main {    public static void main(String[] args) {  String s1 = "ab";  String s2 = "a";  s1 += "c";  s2 += "bc";    System.out.println("s1 = " + s1);  System.out.println("s2 = " + s2);  System.out.println("s1 == s2? " + (s1 == s2));  System.out.println("s1.equals(s2)? " + (s1.equals  (s2)));    }    }    **Quote**  s1 = abc s2 = abc s1 == s2? false s1.equals(s2)? true  However, there is a way for a literal and a constructor value (assuming value is the same) to == each other. [**cfoley**](http://www.dreamincode.net/forums/index.php?s=2d9a2e8043f782095fdefc987235bf18&showuser=61280) introduced me to the intern() method of the String class. What this method does is it looks at its value, and if it matches a value ALREADY IN the String pool, it returns a reference to the object in the pool, else it adds itself to the pool. Observe:  public class Main {    public static void main(String[] args) {  String s1 = "abc";  String s2 = new String("abc");  s2 = s2.intern();    System.out.println("s1 = " + s1);  System.out.println("s2 = " + s2);  System.out.println("s1 == s2? " + (s1 == s2));  System.out.println("s1.equals(s2)? " + (s1.equals  (s2)));    }    }  This returns:  **Quote**  s1 = abc s2 = abc s1 == s2? true s1.equals(s2)? true  Cool huh? This happens because s1 is ASSIGNED to "abc" and is added to the pool. s2, however, is constructed to "abc", but not added to the pool. But the intern() method sees that the VALUE "abc" is already in the pool and thus returns the reference to s1. I got a challenge problem for you now. Will s1 == s2 in the following code?  public class Main {    public static void main(String[] args) {  String s1 = new String("abc");  String s2 = new String("abc");  s2 = s2.intern();    System.out.println("s1 = " + s1);  System.out.println("s2 = " + s2);  System.out.println("s1 == s2? " + (s1 == s2));  System.out.println("s1.equals(s2)? " + (s1.equals  (s2)));    }    }  ... The answer is no. The intern() method is called, but neither s1 nor s2 is in the pool  so all the intern() method does is add it to the pool. To make THESE ==, you must call intern() twice so that there is already a reference in the pool (from the first call).  public class Main {    public static void main(String[] args) {  String s1 = new String("abc");  String s2 = new String("abc");  s2 = s2.intern();  s1 = s1.intern();    System.out.println("s1 = " + s1);  System.out.println("s2 = " + s2);  System.out.println("s1 == s2? " + (s1 == s2));  System.out.println("s1.equals(s2)? " + (s1.equals  (s2)));    }    }    **Quote**  s1 = abc s2 = abc s1 == s2? true s1.equals(s2)? true | | |  | | --- | | **0. StringBuilder and StringBuffer**    :    [**Link**](http://www.techtamasha.com/difference-between-string-and-stringbufferstringbuilder-in-java/28) | | |

## Difference between String and StringBuffer/StringBuilder in Java

Well, the most important difference between String and StringBuffer/StringBuilder in java is that **String object is immutable**whereas **StringBuffer/StringBuilder objects are mutable.**

By immutable, we mean that the value stored in the String object cannot be changed. Then the next question that comes to our mind is “If String is immutable then how am I able to change the contents of the object whenever I wish to?” . Well, to be precise it’s not the same String object that reflects the changes you do. Internally a new String object is created to do the changes.

So suppose you declare a String object:

String myString = “Hello”;

Next, you want to append “Guest” to the same String. What do you do?

myString = myString + ” Guest”;

When you print the contents of myString the output will be “Hello Guest”. Although we made use of the same object(myString), internally a new object was created in the process. So, if you were to do some string operation involving an append or trim or some other method call to modify your string object, you would really be creating those many new objects of class String.

**Now isn’t that a performance issue?**

Yes, it definitely is.

**Then how do you make your string operations efficient?**

By using StringBuffer or StringBuilder.

**How would that help?**

Well, since StringBuffer/StringBuilder objects are mutable, we can make changes to the value stored in the object. What this effectively means is that string operations such as **append** would be more **efficient** if performed using **StringBuffer/StringBuilder** objects than String objects.

**Finally, whats the difference between StringBuffer and StringBuilder?**

StringBuffer and StringBuilder have the same methods with one difference and that’s of synchronization. StringBuffer is synchronized( which means it is thread safe and hence you can use it when you implement threads for your methods) whereas StringBuilder is not synchronized( which implies it isn’t thread safe).

So, if you aren’t going to use threading then use the **StringBuilder** class as it’ll be more **efficient**than **StringBuffer** due to the **absence** of**synchronization**.

[Incase you do not know – Here’s how you use StringBuilder](http://techtamasha.com/?p=30)

[A simple Example to demonstrate that String object is Immutable](http://techtamasha.com/?p=31)

Incase you still have any doubts regarding String or StringBuilder then do leave a comment. I’ll be more than eager to help you out.

**Chapter 14:Generics**

In this tutorial we will learn the concepts of Generics. Use of generics in collection framework and for creating templates

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | |  | | --- | | 0. Use of Generics    :    [Link](http://www.tutorialspoint.com/java/java_generics.htm)  [Link](http://docs.oracle.com/javase/tutorial/java/generics/QandE/generics-questions.html) |   It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array or an array of any type that supports ordering.  Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods or, with a single class declaration, a set of related types, respectively.  Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.  Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements. Generic Methods: You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods:   * All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type ( < E > in the next example). * Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name. * The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments. * A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).  Example: Following example illustrates how we can print array of different type using a single Generic method:  public class GenericMethodTest  {  // generic method printArray  public static < E > void printArray( E[] inputArray )  {  // Display array elements  for ( E element : inputArray ){  System.out.printf( "%s ", element );  }  System.out.println();  }  public static void main( String args[] )  {  // Create arrays of Integer, Double and Character  Integer[] intArray = { 1, 2, 3, 4, 5 };  Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };  Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };  System.out.println( "Array integerArray contains:" );  printArray( intArray ); // pass an Integer array  System.out.println( "\nArray doubleArray contains:" );  printArray( doubleArray ); // pass a Double array  System.out.println( "\nArray characterArray contains:" );  printArray( charArray ); // pass a Character array  }  }  This would produce the following result:  Array integerArray contains:  1 2 3 4 5 6  Array doubleArray contains:  1.1 2.2 3.3 4.4  Array characterArray contains:  H E L L O Bounded Type Parameters: There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.  To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound. Example: Following example illustrates how extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces). This example is Generic method to return the largest of three Comparable objects:  public class MaximumTest  {  // determines the largest of three Comparable objects  public static <T extends Comparable<T>> T maximum(T x, T y, T z)  {  T max = x; // assume x is initially the largest  if ( y.compareTo( max ) > 0 ){  max = y; // y is the largest so far  }  if ( z.compareTo( max ) > 0 ){  max = z; // z is the largest now  }  return max; // returns the largest object  }  public static void main( String args[] )  {  System.out.printf( "Max of %d, %d and %d is %d\n\n",  3, 4, 5, maximum( 3, 4, 5 ) );  System.out.printf( "Maxm of %.1f,%.1f and %.1f is %.1f\n\n",  6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );  System.out.printf( "Max of %s, %s and %s is %s\n","pear",  "apple", "orange", maximum( "pear", "apple", "orange" ) );  }  }  This would produce the following result:  Maximum of 3, 4 and 5 is 5  Maximum of 6.6, 8.8 and 7.7 is 8.8  Maximum of pear, apple and orange is pear Generic Classes: A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.  As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters. Example: Following example illustrates how we can define a generic class:  public class Box<T> {  private T t;  public void add(T t) {  this.t = t;  }  public T get() {  return t;  }  public static void main(String[] args) {  Box<Integer> integerBox = new Box<Integer>();  Box<String> stringBox = new Box<String>();    integerBox.add(new Integer(10));  stringBox.add(new String("Hello World"));  System.out.printf("Integer Value :%d\n\n", integerBox.get());  System.out.printf("String Value :%s\n", stringBox.get());  }  }  This would produce the following result:  Integer Value :10  String Value :Hello World Questions and Exercises: Generics  1. Write a generic method to count the number of elements in a collection that have a specific property (for example, odd integers, prime numbers, palindromes). 2. Will the following class compile? If not, why? 3. public final class Algorithm { 4. public static <T> T max(T x, T y) { 5. return x > y ? x : y; 6. } 7. } 8. Write a generic method to exchange the positions of two different elements in an array. 9. If the compiler erases all type parameters at compile time, why should you use generics? 10. What is the following class converted to after type erasure? 11. public class Pair<K, V> { 12. public Pair(K key, V value) { 13. this.key = key; 14. this.value = value; 15. } 16. public K getKey(); { return key; } 17. public V getValue(); { return value; } 18. public void setKey(K key) { this.key = key; } 19. public void setValue(V value) { this.value = value; } 20. private K key; 21. private V value; 22. } 23. What is the following method converted to after type erasure? 24. public static <T extends Comparable<T>> 25. int findFirstGreaterThan(T[] at, T elem) { 26. // ... 27. } 28. Will the following method compile? If not, why? 29. public static void print(List<? extends Number> list) { 30. for (Number n : list) 31. System.out.print(n + " "); 32. System.out.println(); 33. } 34. Write a generic method to find the maximal element in the range [begin, end) of a list. 35. Will the following class compile? If not, why? 36. public class Singleton<T> { 37. public static T getInstance() { 38. if (instance == null) 39. instance = new Singleton<T>(); 40. return instance; 41. } 42. private static T instance = null; 43. } 44. Given the following classes: 45. class Shape { /\* ... \*/ } 46. class Circle extends Shape { /\* ... \*/ } 47. class Rectangle extends Shape { /\* ... \*/ } 48. class Node<T> { /\* ... \*/ }   Will the following code compile? If not, why?  Node<Circle> nc = new Node<>();  Node<Shape> ns = nc;   1. Consider this class: 2. class Node<T> implements Comparable<T> { 3. public int compareTo(T obj) { /\* ... \*/ } 4. // ... 5. }   Will the following code compile? If not, why?  Node<String> node = new Node<>();  Comparable<String> comp = node;   1. How do you invoke the following method to find the first integer in a list that is relatively prime to a list of specified integers? 2. public static <T> 3. int findFirst(List<T> list, int begin, int end, UnaryPredicate<T> p)   Note that two integers a and b are relatively prime if gcd(a, b) = 1, where gcd is short for greatest common divisor.  Answers: Answer to Questions and Exercises: Generics  1. Write a generic method to count the number of elements in a collection that have a specific property (for example, odd integers, prime numbers, palindromes).  **Answer**: 2. public final class Algorithm { 3. public static <T> int countIf(Collection<T> c, UnaryPredicate<T> p) { 4. int count = 0; 5. for (T elem : c) 6. if (p.test(elem)) 7. ++count; 8. return count; 9. } 10. }   where the generic UnaryPredicate interface is defined as follows:  public interface UnaryPredicate<T> {  public boolean test(T obj);  }  For example, the following program counts the number of odd integers in an integer list:  import java.util.\*;  class OddPredicate implements UnaryPredicate<Integer> {  public boolean test(Integer i) { return i % 2 != 0; }  }  public class Test {  public static void main(String[] args) {  Collection<Integer> ci = Arrays.asList(1, 2, 3, 4);  int count = Algorithm.countIf(ci, new OddPredicate());  System.out.println("Number of odd integers = " + count);  }  }  The program prints:  Number of odd integers = 2   1. Will the following class compile? If not, why? 2. public final class Algorithm { 3. public static <T> T max(T x, T y) { 4. return x > y ? x : y; 5. } 6. }   **Answer**: No. The greater than (>) operator applies only to primitive numeric types.   1. Write a generic method to exchange the positions of two different elements in an array.  **Answer**: 2. public final class Algorithm { 3. public static <T> void swap(T[] a, int i, int j) { 4. T temp = a[i]; 5. a[i] = a[j]; 6. a[j] = temp; 7. } 8. } 9. If the compiler erases all type parameters at compile time, why should you use generics?  **Answer**: You should use generics because:    * The Java compiler enforces tighter type checks on generic code at compile time.    * Generics support programming types as parameters.    * Generics enable you to implement generic algorithms. 10. What is the following class converted to after type erasure? 11. public class Pair<K, V> { 12. public Pair(K key, V value) { 13. this.key = key; 14. this.value = value; 15. } 16. public K getKey(); { return key; } 17. public V getValue(); { return value; } 18. public void setKey(K key) { this.key = key; } 19. public void setValue(V value) { this.value = value; } 20. private K key; 21. private V value; 22. }   **Answer**:  public class Pair {  public Pair(Object key, Object value) {  this.key = key;  this.value = value;  }  public Object getKey() { return key; }  public Object getValue() { return value; }  public void setKey(Object key) { this.key = key; }  public void setValue(Object value) { this.value = value; }  private Object key;  private Object value;  }   1. What is the following method converted to after type erasure? 2. public static <T extends Comparable<T>> 3. int findFirstGreaterThan(T[] at, T elem) { 4. // ... 5. }   **Answer**:  public static int findFirstGreaterThan(Comparable[] at, Comparable elem) {  // ...  }   1. Will the following method compile? If not, why? 2. public static void print(List<? extends Number> list) { 3. for (Number n : list) 4. System.out.print(n + " "); 5. System.out.println(); 6. }   **Answer**: Yes.   1. Write a generic method to find the maximal element in the range [begin, end) of a list.  **Answer**: 2. import java.util.\*; 3. public final class Algorithm { 4. public static <T extends Object & Comparable<? super T>> 5. T max(List<? extends T> list, int begin, int end) { 6. T maxElem = list.get(begin); 7. for (++begin; begin < end; ++begin) 8. if (maxElem.compareTo(list.get(begin)) < 0) 9. maxElem = list.get(begin); 10. return maxElem; 11. } 12. } 13. Will the following class compile? If not, why? 14. public class Singleton<T> { 15. public static T getInstance() { 16. if (instance == null) 17. instance = new Singleton<T>(); 18. return instance; 19. } 20. private static T instance = null; 21. }   **Answer**: No. You cannot create a static field of the type parameter T.   1. Given the following classes: 2. class Shape { /\* ... \*/ } 3. class Circle extends Shape { /\* ... \*/ } 4. class Rectangle extends Shape { /\* ... \*/ } 5. class Node<T> { /\* ... \*/ }   Will the following code compile? If not, why?  Node<Circle> nc = new Node<>();  Node<Shape> ns = nc;  **Answer**: No. Because Node<Circle> is not a subtype of Node<Shape>.   1. Consider this class: 2. class Node<T> implements Comparable<T> { 3. public int compareTo(T obj) { /\* ... \*/ } 4. // ... 5. }   Will the following code compile? If not, why?  **Answer**: Yes.  Node<String> node = new Node<>();  Comparable<String> comp = node;   1. How do you invoke the following method to find the first integer in a list that is relatively prime to a list of specified integers? 2. public static <T> 3. int findFirst(List<T> list, int begin, int end, UnaryPredicate<T> p)   Note that two integers a and b are relatively prime if gcd(a, b) = 1, where gcd is short for greatest common divisor.  **Answer**:  import java.util.\*;  public final class Algorithm {  public static <T>  int findFirst(List<T> list, int begin, int end, UnaryPredicate<T> p) {  for (; begin < end; ++begin)  if (p.test(list.get(begin)))  return begin;  return -1;  }  // x > 0 and y > 0  public static int gcd(int x, int y) {  for (int r; (r = x % y) != 0; x = y, y = r) { }  return y;  }  }  The generic UnaryPredicate interface is defined as follows:  public interface UnaryPredicate<T> {  public boolean test(T obj);  }  The following program tests the findFirst method:  import java.util.\*;  class RelativelyPrimePredicate implements UnaryPredicate<Integer> {  public RelativelyPrimePredicate(Collection<Integer> c) {  this.c = c;  }  public boolean test(Integer x) {  for (Integer i : c)  if (Algorithm.gcd(x, i) != 1)  return false;  return c.size() > 0;  }  private Collection<Integer> c;  }  public class Test {  public static void main(String[] args) throws Exception {  List<Integer> li = Arrays.asList(3, 4, 6, 8, 11, 15, 28, 32);  Collection<Integer> c = Arrays.asList(7, 18, 19, 25);  UnaryPredicate<Integer> p = new RelativelyPrimePredicate(c);  int i = ALgorithm.findFirst(li, 0, li.size(), p);  if (i != -1) {  System.out.print(li.get(i) + " is relatively prime to ");  for (Integer k : c)  System.out.print(k + " ");  System.out.println();  }  }  }  The program prints:  11 is relatively prime to 7 18 19 25 | | |  | | --- | | 0. WildCard Instantiation    :    [Link](http://docs.oracle.com/javase/tutorial/extra/generics/wildcards.html) |   Consider the problem of writing a routine that prints out all the elements in a collection. Here's how you might write it in an older version of the language (i.e., a pre-5.0 release):  void printCollection(Collection c) {  Iterator i = c.iterator();  for (k = 0; k < c.size(); k++) {  System.out.println(i.next());  }  }  And here is a naive attempt at writing it using generics (and the new for loop syntax):  void printCollection(Collection<Object> c) {  for (Object e : c) {  System.out.println(e);  }  }  The problem is that this new version is much less useful than the old one. Whereas the old code could be called with any kind of collection as a parameter, the new code only takes Collection<Object>, which, as we've just demonstrated, is not a supertype of all kinds of collections!  So what is the supertype of all kinds of collections? It's written Collection<?> (pronounced "collection of unknown"), that is, a collection whose element type matches anything. It's called a wildcard type for obvious reasons. We can write:  void printCollection(Collection<?> c) {  for (Object e : c) {  System.out.println(e);  }  }  and now, we can call it with any type of collection. Notice that inside printCollection(), we can still read elements from c and give them type Object. This is always safe, since whatever the actual type of the collection, it does contain objects. It isn't safe to add arbitrary objects to it however:  Collection<?> c = new ArrayList<String>();  c.add(new Object()); // Compile time error  Since we don't know what the element type of c stands for, we cannot add objects to it. The add() method takes arguments of type E, the element type of the collection. When the actual type parameter is ?, it stands for some unknown type. Any parameter we pass to add would have to be a subtype of this unknown type. Since we don't know what type that is, we cannot pass anything in. The sole exception is null, which is a member of every type.  On the other hand, given a List<?>, we can call get() and make use of the result. The result type is an unknown type, but we always know that it is an object. It is therefore safe to assign the result of get() to a variable of type Object or pass it as a parameter where the type Object is expected. Bounded Wildcards Consider a simple drawing application that can draw shapes such as rectangles and circles. To represent these shapes within the program, you could define a class hierarchy such as this:  public abstract class Shape {  public abstract void draw(Canvas c);  }  public class Circle extends Shape {  private int x, y, radius;  public void draw(Canvas c) {  ...  }  }  public class Rectangle extends Shape {  private int x, y, width, height;  public void draw(Canvas c) {  ...  }  }  These classes can be drawn on a canvas:  public class Canvas {  public void draw(Shape s) {  s.draw(this);  }  }  Any drawing will typically contain a number of shapes. Assuming that they are represented as a list, it would be convenient to have a method in Canvas that draws them all:  public void drawAll(List<Shape> shapes) {  for (Shape s: shapes) {  s.draw(this);  }  }  Now, the type rules say that drawAll() can only be called on lists of exactly Shape: it cannot, for instance, be called on a List<Circle>. That is unfortunate, since all the method does is read shapes from the list, so it could just as well be called on a List<Circle>. What we really want is for the method to accept a list of any kind of shape:  public void drawAll(List<? extends Shape> shapes) {  ...  }  There is a small but very important difference here: we have replaced the type List<Shape> with List<? extends Shape>. Now drawAll() will accept lists of any subclass of Shape, so we can now call it on a List<Circle> if we want.  List<? extends Shape> is an example of a *bounded wildcard*. The ? stands for an unknown type, just like the wildcards we saw earlier. However, in this case, we know that this unknown type is in fact a subtype of Shape. (Note: It could be Shape itself, or some subclass; it need not literally extend Shape.) We say that Shape is the *upper bound* of the wildcard.  There is, as usual, a price to be paid for the flexibility of using wildcards. That price is that it is now illegal to write into shapes in the body of the method. For instance, this is not allowed:  public void addRectangle(List<? extends Shape> shapes) {  // *Compile-time error!*  shapes.add(0, new Rectangle());  }  You should be able to figure out why the code above is disallowed. The type of the second parameter to shapes.add() is ? extends Shape-- an unknown subtype ofShape. Since we don't know what type it is, we don't know if it is a supertype of Rectangle; it might or might not be such a supertype, so it isn't safe to pass a Rectanglethere.  Bounded wildcards are just what one needs to handle the example of the DMV passing its data to the census bureau. Our example assumes that the data is represented by mapping from names (represented as strings) to people (represented by reference types such as Person or its subtypes, such as Driver). Map<K,V> is an example of a generic type that takes two type arguments, representing the keys and values of the map.  Again, note the naming convention for formal type parameters--K for keys and V for values.  public class Census {  public static void addRegistry(Map<String, ? extends Person> registry) {  }  ...  Map<String, Driver> allDrivers = ... ;  Census.addRegistry(allDrivers); | | |  | | --- | | 0. Introduction to template classes    :    [Link](http://docs.oracle.com/javase/tutorial/java/generics/types.html)  [Link](http://www.cs.washington.edu/education/courses/cse341/06au/java/exercises.pdf) |  Generic Types A generic type is a generic class or interface that is parameterized over types. The following Box class will be modified to demonstrate the concept. A Simple Box Class Begin by examining a non-generic Box class that operates on objects of any type. It needs only to provide two methods: set, which adds an object to the box, and get, which retrieves it:  public class Box {  private Object object;  public void set(Object object) { this.object = object; }  public Object get() { return object; }  }  Since its methods accept or return an Object, you are free to pass in whatever you want, provided that it is not one of the primitive types. There is no way to verify, at compile time, how the class is used. One part of the code may place an Integer in the box and expect to get Integers out of it, while another part of the code may mistakenly pass in aString, resulting in a runtime error. A Generic Version of the Box Class A generic class is defined with the following format:  class name<T1, T2, ..., Tn> { /\* ... \*/ }  The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the type parameters (also called type variables) T1, T2, ..., and Tn.  To update the Box class to use generics, you create a generic type declaration by changing the code "public class Box" to "public class Box<T>". This introduces the type variable,T, that can be used anywhere inside the class.  With this change, the Box class becomes:  /\*\*  \* Generic version of the Box class.  \* @param <T> the type of the value being boxed  \*/  public class Box<T> {  // T stands for "Type"  private T t;  public void set(T t) { this.t = t; }  public T get() { return t; }  }  As you can see, all occurrences of Object are replaced by T. A type variable can be any **non-primitive** type you specify: any class type, any interface type, any array type, or even another type variable.  This same technique can be applied to create generic interfaces. Type Parameter Naming Conventions By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable [naming](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html#naming) conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.  The most commonly used type parameter names are:   * E - Element (used extensively by the Java Collections Framework) * K - Key * N - Number * T - Type * V - Value * S,U,V etc. - 2nd, 3rd, 4th types   You'll see these names used throughout the Java SE API and the rest of this lesson. Invoking and Instantiating a Generic Type To reference the generic Box class from within your code, you must perform a generic type invocation, which replaces T with some concrete value, such as Integer:  Box<Integer> integerBox;  You can think of a generic type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a type argument —Integer in this case — to the Box class itself.  **Type Parameter and Type Argument Terminology:** Many developers use the terms "type parameter" and "type argument" interchangeably, but these terms are not the same. When coding, one provides type arguments in order to create a parameterized type. Therefore, the T in Foo<T> is a type parameter and the String inFoo<String> f is a type argument. This lesson observes this definition when using these terms.  Like any other variable declaration, this code does not actually create a new Box object. It simply declares that integerBox will hold a reference to a "Box of Integer", which is howBox<Integer> is read.  An invocation of a generic type is generally known as a parameterized type.  To instantiate this class, use the new keyword, as usual, but place <Integer> between the class name and the parenthesis:  Box<Integer> integerBox = new Box<Integer>(); The Diamond In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, <>, is informally called the diamond. For example, you can create an instance ofBox<Integer> with the following statement:  Box<Integer> integerBox = new Box<>();  For more information on diamond notation and type inference, see [Type Inference](http://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html). Multiple Type Parameters As mentioned previously, a generic class can have multiple type parameters. For example, the generic OrderedPair class, which implements the generic Pair interface:  public interface Pair<K, V> {  public K getKey();  public V getValue();  }  public class OrderedPair<K, V> implements Pair<K, V> {  private K key;  private V value;  public OrderedPair(K key, V value) {  this.key = key;  this.value = value;  }  public K getKey() { return key; }  public V getValue() { return value; }  }  The following statements create two instantiations of the OrderedPair class:  Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);  Pair<String, String> p2 = new OrderedPair<String, String>("hello", "world");  The code, new OrderedPair<String, Integer>, instantiates K as a String and V as an Integer. Therefore, the parameter types of OrderedPair's constructor are String and Integer, respectively. Due to [autoboxing](http://docs.oracle.com/javase/tutorial/java/data/autoboxing.html), it is valid to pass a String and an int to the class.  As mentioned in [The Diamond](http://docs.oracle.com/javase/tutorial/java/generics/types.html#diamond), because a Java compiler can infer the K and V types from the declaration OrderedPair<String, Integer>, these statements can be shortened using diamond notation:  OrderedPair<String, Integer> p1 = new OrderedPair**<>**("Even", 8);  OrderedPair<String, String> p2 = new OrderedPair**<>**("hello", "world");  To create a generic interface, follow the same conventions as for creating a generic class. Parameterized Types You can also substitute a type parameter (i.e., K or V) with a parameterized type (i.e., List<String>). For example, using the OrderedPair<K, V> example:  OrderedPair<String, **Box<Integer>**> p = new OrderedPair<>("primes", new Box<Integer>(...)); | | |  | | --- | | 0. Generic Methods    :    [Link](http://docs.oracle.com/javase/tutorial/extra/generics/methods.html) | | |  |

# Generic Methods

Consider writing a method that takes an array of objects and a collection and puts all objects in the array into the collection. Here's a first attempt:

**static void** fromArrayToCollection(Object[] a, Collection<?> c) {

**for** (Object o : a) {

c.add(o); // *compile-time error*

}

}

By now, you will have learned to avoid the beginner's mistake of trying to use Collection<Object> as the type of the collection parameter. You may or may not have recognized that using Collection<?> isn't going to work either. Recall that you cannot just shove objects into a collection of unknown type.

The way to do deal with these problems is to use *generic methods*. Just like type declarations, method declarations can be generic—that is, parameterized by one or more type parameters.

**static** <T> **void** fromArrayToCollection(T[] a, Collection<T> c) {

**for** (T o : a) {

c.add(o); // *Correct*

}

}

We can call this method with any kind of collection whose element type is a supertype of the element type of the array.

Object[] oa = new Object[100];

Collection<Object> co = new ArrayList<Object>();

// *T inferred to be Object*

fromArrayToCollection(oa, co);

String[] sa = new String[100];

Collection<String> cs = new ArrayList<String>();

// *T inferred to be String*

fromArrayToCollection(sa, cs);

// *T inferred to be Object*

fromArrayToCollection(sa, co);

Integer[] ia = new Integer[100];

Float[] fa = new Float[100];

Number[] na = new Number[100];

Collection<Number> cn = new ArrayList<Number>();

// *T inferred to be Number*

fromArrayToCollection(ia, cn);

// *T inferred to be Number*

fromArrayToCollection(fa, cn);

// *T inferred to be Number*

fromArrayToCollection(na, cn);

// *T inferred to be Object*

fromArrayToCollection(na, co);

// *compile-time error*

fromArrayToCollection(na, cs);

Notice that we don't have to pass an actual type argument to a generic method. The compiler infers the type argument for us, based on the types of the actual arguments. It will generally infer the most specific type argument that will make the call type-correct.

One question that arises is: when should I use generic methods, and when should I use wildcard types? To understand the answer, let's examine a few methods from theCollection libraries.

**interface** Collection<E> {

**public boolean** containsAll(Collection<?> c);

**public boolean** addAll(Collection<? **extends E**> c);

}

We could have used generic methods here instead:

**interface** Collection<E> {

**public** <T> **boolean** containsAll(Collection<T> c);

**public** <T **extends** E> **boolean** addAll(Collection<T> c);

// *Hey, type variables can have bounds too!*

}

However, in both containsAll and addAll, the type parameter T is used only once. The return type doesn't depend on the type parameter, nor does any other argument to the method (in this case, there simply is only one argument). This tells us that the type argument is being used for polymorphism; its only effect is to allow a variety of actual argument types to be used at different invocation sites. If that is the case, one should use wildcards. Wildcards are designed to support flexible subtyping, which is what we're trying to express here.

Generic methods allow type parameters to be used to express dependencies among the types of one or more arguments to a method and/or its return type. If there isn't such a dependency, a generic method should not be used.

It is possible to use both generic methods and wildcards in tandem. Here is the method Collections.copy():

**class** Collections {

**public static** <T> **void** copy(List<T> dest, List<? **extends** T> src) {

...

}

Note the dependency between the types of the two parameters. Any object copied from the source list, src, must be assignable to the element type T of the destination list, dst. So the element type of src can be any subtype of T—we don't care which. The signature of copy expresses the dependency using a type parameter, but uses a wildcard for the element type of the second parameter.

We could have written the signature for this method another way, without using wildcards at all:

**class** Collections {

**public static** <T, S **extends** T> **void** copy(List<T> dest, List<S> src) {

...

}

This is fine, but while the first type parameter is used both in the type of dst and in the bound of the second type parameter, S, S itself is only used once, in the type of src—nothing else depends on it. This is a sign that we can replace S with a wildcard. Using wildcards is clearer and more concise than declaring explicit type parameters, and should therefore be preferred whenever possible.

Wildcards also have the advantage that they can be used outside of method signatures, as the types of fields, local variables and arrays. Here is an example.

Returning to our shape drawing problem, suppose we want to keep a history of drawing requests. We can maintain the history in a static variable inside class Shape, and havedrawAll() store its incoming argument into the history field.

**static** List<List<? extends Shape>>

history = new ArrayList<List<? extends Shape>>();

**public void** drawAll(List<? **extends** Shape> shapes) {

history.addLast(shapes);

**for** (Shape s: shapes) {

s.draw(**this**);

}

}

Finally, again let's take note of the naming convention used for the type parameters. We use T for type, whenever there isn't anything more specific about the type to distinguish it. This is often the case in generic methods. If there are multiple type parameters, we might use letters that neighbor T in the alphabet, such as S. If a generic method appears inside a generic class, it's a good idea to avoid using the same names for the type parameters of the method and class, to avoid confusion. The same applies to nested generic classes.

**Chapter 15:Exception Handling and Assertions**

An exception is a problem that arises during the execution of a program due to multiple reasons like a user has entered invalid value, a file that needs to be opened cannot be found, a network connection has been lost in the middle of communications, etc... In this tutorial we will see how to handle such situations and avoid unexpected termination of the code.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | |  | | --- | | **0. Legacy error handling vs. exception handling**    :    [**Link**](http://www.javatpoint.com/exception-handling-and-checked-and-unchecked-exception)  [**Link**](https://eis.persistent.co.in/eis/candidateportal/webui/Campus/www.homeworktutoring.com/samples/Java%20assignment%202.doc%e2%80%8e) | | | |  | | --- | | **0. try-catch-finally blocks**    :    [**Link**](http://www.javatpoint.com/try-catch-block)  [**Link**](http://www.programr.com/practice/programr211_promo_228/node/543) |  Java try-catchJava try block Java try block is used to enclose the code that might throw an exception. It must be used within the method.  Java try block must be followed by either catch or finally block. Syntax of java try-catch  1. **try**{ 2. //code that may throw exception 3. }**catch**(Exception\_class\_Name ref){}  Syntax of try-finally block  1. **try**{ 2. //code that may throw exception 3. }**finally**{}  Java catch block Java catch block is used to handle the Exception. It must be used after the try block only.  You can use multiple catch block with a single try. Problem without exception handling Let's try to understand the problem if we don't use try-catch block.   1. **public** **class** Testtrycatch1{ 2. **public** **static** **void** main(String args[]){ 3. **int** data=50/0;//may throw exception 4. System.out.println("rest of the code..."); 5. } 6. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testtrycatch1)  Output:  Exception in thread main java.lang.ArithmeticException:/ by zero  As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).  There can be 100 lines of code after exception. So all the code after exception will not be executed. Solution by exception handling Let's see the solution of above problem by java try-catch block.   1. **public** **class** Testtrycatch2{ 2. **public** **static** **void** main(String args[]){ 3. **try**{ 4. **int** data=50/0; 5. }**catch**(ArithmeticException e){System.out.println(e);} 6. System.out.println("rest of the code..."); 7. } 8. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testtrycatch2)  Output:  Exception in thread main java.lang.ArithmeticException:/ by zero  rest of the code...  Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed. Internal working of java try-catch block internal working of try-catch block  The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:   * Prints out exception description. * Prints the stack trace (Hierarchy of methods where the exception occurred). * Causes the program to terminate.   But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed. | | |  | | --- | | **0. Checked and Unchecked exceptions**    :    [**Link**](http://tutorials.jenkov.com/java-exception-handling/checked-or-unchecked-exceptions.html)  [**Link**](http://www.programr.com/practice/programr211_promo_228/node/570) |  Checked or Unchecked Exceptions? http://tutorials.jenkov.com/images/layout/small-portrait_mini.jpg  By [Jakob Jenkov](http://jakob.jenkov.com/)   Connect with me:  Rate article:  Share article:    Table of Contents   * [A Simple Example](http://tutorials.jenkov.com/java-exception-handling/checked-or-unchecked-exceptions.html#a-simple-example) * [Checked or Unchecked Exceptions?](http://tutorials.jenkov.com/java-exception-handling/checked-or-unchecked-exceptions.html#checked-or-unchecked-exceptions) * [Summary](http://tutorials.jenkov.com/java-exception-handling/checked-or-unchecked-exceptions.html#summary)   In Java there are basically two types of exceptions: Checked exceptions and unchecked exceptions. C# only has unchecked exceptions. The differences between checked and unchecked exceptions are:   1. Checked exceptions must be explicitly caught or propagated as described in [**Basic try-catch-finally Exception Handling**](http://tutorials.jenkov.com/java-exception-handling/basic-try-catch-finally.html). Unchecked exceptions do not have this requirement. They don't have to be caught or declared thrown. 2. Checked exceptions in Java extend the java.lang.Exception class. Unchecked exceptions extend the java.lang.RuntimeException.   There are many arguments for and against both checked and unchecked, and whether to use checked exceptions at all. I will go through the most common arguments throughout this text. Before I do so, let me just make one thing clear:  Checked and unchecked exceptions are functionally equivalent. There is nothing you can do with checked exceptions that cannot also be done with unchecked exceptions, and vice versa.  Regardless of your choice between checked and unchecked exceptions it is a matter of personal or organisational style. None is functionally better than the other. A Simple Example Before discussing the advantages and disadvantages of checked and unchecked exceptions I will show you the difference in the code they make. Here is a method that throws a checked exception, and another method that calls it:  public void storeDataFromUrl(String url){  try {  String data = readDataFromUrl(url);  } catch (BadUrlException e) {  e.printStackTrace();  }  }  public String readDataFromUrl(String url)  throws BadUrlException{  if(isUrlBad(url)){  throw new BadUrlException("Bad URL: " + url);  }  String data = null;  //read lots of data over HTTP and return  //it as a String instance.  return data;  }  As you can see the readDataFromUrl() method throws a BadUrlException. I have created BadUrlException myself. BadUrlException is a checked exception because it extends java.lang.Exception:  public class BadUrlException extends Exception {  public BadUrlException(String s) {  super(s);  }  }  If storeDataFromUrl() wants to call readDataFromUrl() it has only two choices. Either it catches the BadUrlException or propagates it up the call stack. The storeDataFromUrl() listed above catches the exception. This storeDataFromUrl() implementation propagates the BadUrlException instead:  public void storeDataFromUrl(String url)  throws BadUrlException{  String data = readDataFromUrl(url);  }  Notice how the try catch block is gone and a "throws BadUrlException" declaration is added instead. Now, let's see how it looks with unchecked exceptions. First I change the BadUrlException to extend java.lang.RuntimeException instead:  public class BadUrlException extends RuntimeException {  public BadUrlException(String s) {  super(s);  }  }  Then I change the methods to use the now unchecked BadUrlException:  public void storeDataFromUrl(String url){  String data = readDataFromUrl(url);  }  public String readDataFromUrl(String url) {  if(isUrlBad(url)){  throw new BadUrlException("Bad URL: " + url);  }  String data = null;  //read lots of data over HTTP and  //return it as a String instance.  return data;  }  Notice how the readDataFromUrl() method no longer declares that it throws BadUrlException. The storeDataFromUrl() method doesn't have to catch the BadUrlException either. The storeDataFromUrl() method can still choose to catch the exception but it no longer has to, and it no longer has to declare that it propagates the exception. Checked or Unchecked Exceptions? Now that we have seen the difference in code between checked and unchecked exceptions, let's dive into the arguments for and against both.  Some Java books(\*) covering exceptions advice you to use checked exceptions for all errors the application can recover from, and unchecked exceptions for the errors the application cannot recover from. In reality most applications will have to recover from pretty much all exceptions including NullPointerException, IllegalArgumentExceptions and many other unchecked exceptions. The action / transaction that failed will be aborted but the application has to stay alive and be ready to serve the next action / transaction. The only time it is normally legal to shut down an application is during startup. For instance, if a configuration file is missing and the application cannot do anything sensible without it, then it is legal to shut down the application.  (\*) Suns Java Tutorial does for one.  My advice to you is to use either only checked exceptions or only unchecked exceptions. Mixing exception types often results in confusion and inconsistent use. Of course you should be pragmatic. Do what makes sense in your situation.  Below is a list of the most common arguments for and against checked and unchecked exceptions. An argument in favor of one type of exceptions is usually against the other type (pro-checked = con-unchecked, pro-unchecked = con-checked). Therefore the arguments are only listed as either in favour of checked or unchecked exceptions.   1. Pro Checked Exceptions:  Compiler enforced catching or propagation of checked exceptions make it harder to forget handling that exception. 2. Pro Checked Exceptions:  Unchecked exceptions makes it easier to forget handling errors since the compiler doesn't force the developer to catch or propagate exceptions (reverse of 1). 3. Pro Unchecked Exceptions:  Checked exceptions that are propagated up the call stack clutter the top level methods, because these methods need to declare throwing all exceptions thrown from methods they call. 4. Pro Checked Exceptions: When methods do not declare what unchecked exceptions they may throw it becomes more difficult to handle them. 5. Pro Unchecked Exceptions: Checked exceptions thrown become part of a methods interface and makes it harder to add or remove exceptions from the method in later versions of the class or interface.   Each of the arguments also have counter arguments which will be discussed as I go through the argument in the following sections. Argument 1 (Pro Checked Exceptions): Compiler enforced catching or propagation of checked exceptions makes it harder to forget the handling of that exception. Counter-argument: When being forced to catch or propagate many exceptions developers risk acting sloppily, and just write  try{  callMethodThatThrowsException();  catch(Exception e){  }  and thus effectively ignore the error.  Argument 2 (Pro Checked Exceptions): Unchecked exceptions makes it easier to forget handling errors since the compiler doesn't force the developer to catch or propagate exceptions. Counter-argument 1: It's not any worse than the sloppy exception handling tendency when being forced to handle or propagate checked exceptions. Counter-argument 2: On a recent larger project we decided to go with unchecked exceptions. My personal experience from that project is this: When using unchecked exceptions any method can potentially throw exceptions. Thus I was always reasonably conscious about exceptions no matter what parts of the code I was working on. Not only when checked exceptions were declared.  In addition many of the standard Java API methods that do not declare any checked exceptions may still throw unchecked exceptions like NullPointerException or InvalidArgumentException. Your application will still need to handle these unchecked exceptions. You could argue that the fact that there are checked exceptions makes it easy to forget handling the unchecked exceptions because they are not declared.  Argument 3 (Pro Unchecked Exceptions): Checked exceptions that are propagated up the call stack clutter the top level methods, because these methods need to declare throwing all exceptions thrown from methods they call. That is. the declared exceptions are aggreated up the methods in the call stack. Example:  public long readNumberFromUrl(String url)  throws BadUrlExceptions, BadNumberException{  String data = readDataFromUrl(url);  long number = convertData(data);  return number;  }  private String readDataFromUrl(String url)  throws BadUrlException {  //throw BadUrlException if url is bad.  //read data and return it.  }  private long convertData(String data)  throws BadNumberException{  //convert data to long.  //throw BadNumberException if number isn't within valid range.  }  As you can see the readNumberFromUrl() needs to declare throwing both the BadUrlException and the BadNumberException that are thrown from the readDataFromUrl() and converData() methods. Imagine how many exceptions would need to be declared at the top level methods of an application with thousands of classes. This can make checked exception propagation a real pain. Counter-argument 1: The exception declaration aggregation rarely happens in real applications. Often developers will use exception wrapping instead. Here is how that could look:  public void readNumberFromUrl(String url)  throws ApplicationException{  try{  String data = readDataFromUrl(url);  long number = convertData(data);  } catch (BadUrlException e){  throw new ApplicationException(e);  } catch (BadNumberException e){  throw new ApplicationException(e);  }  }  As you can see the readNumberFromUrl() method now only declares throwing ApplicationException. The exceptions BadUrlException and BadNumberException are caught and wrapped in a more general ApplicationException. This way exception wrapping avoids exception declaration aggregation.  My personal opinion is, that if all you do is to wrap the exception and not provide any extra information, why wrap it at all? The try-catch block is just extra code that doesn't do anything. It would be easier to just make the ApplicationException, BadUrlException and BadNumberException be unchecked exceptions. Here is an unchecked version of the above code:  public void readNumberFromUrl(String url){  String data = readDataFromUrl(url);  long number = convertData(data);  }  It is still possible to wrap unchecked exceptions if you should want to. Below is a wrapping edition of the unchecked code. Notice how the readNumberFromUrl() method does not declare throwing the ApplicationException even if it throws it.  public void readNumberFromUrl(String url)  try{  String data = readDataFromUrl(url);  long number = convertData(data);  } catch (BadUrlException e){  throw new ApplicationException(  "Error reading number from URL", e);  } catch (BadNumberException e){  throw new ApplicationException(  "Error reading number from URL", e);  }  } Counter-argument 2: Another commonly used technique to avoid exception declaration aggregation up the call stack of an application is to create an application base exception. All exceptions thrown in the application must be a subclass of the base exception. All methods throwing exceptions need only declare to throw the base exception. As you know a method throwing Exception may also throw any subclass of Exception. Here is how that could look:  public long readNumberFromUrl(String url)  throws ApplicationException {  String data = readDataFromUrl(url);  long number = convertData(data);  return number;  }  private String readDataFromUrl(String url)  throws BadUrlException {  //throw BadUrlException if url is bad.  //read data and return it.  }  private long convertData(String data)  throws BadNumberException{  //convert data to long.  //throw BadNumberException if number isn't within valid range.  }  public class ApplicationException extends Exception{ }  public class BadNumberException extends ApplicationException{}  public class BadUrlException extends ApplicationException{}  Notice how the BadNumberException and BadUrlException are no longer declared thrown nor caught and wrapped. They are subclasses of the ApplicationException so they will get propagated up the call stack.  My opinion is the same as with exception wrapping: If all methods in the application just declares throwing the ApplicationException (base exception), why not just make the ApplicationException unchecked and save some try-catch blocks and throws ApplicationExceptions clauses? Argument 4 (Pro Checked Exceptions) When methods do not declare what unchecked exceptions they may throw it becomes more difficult to handle them. Without declaration you cannot know which exceptions the method may throw. Thus you may not know how to handle them properly. Except of course, if you have access to the code and can see there what exceptions may be thrown from the method. Counter-argument: In most cases you cannot do anything about the exception except showing an error message to the user, write a message to the log, and/or rollback the transaction etc. No matter what exception occurs you will in many situations handle it the same way. Because of this applications often have a few central and general pieces of error handling code. Therefore it is not so important to know exactly what exceptions may be thrown. Argument 5 (Pro Unchecked Exceptions) Checked exceptions declared on methods become part of a the class or interface contract. This makes it harder to add new exceptions to the method later without breaking the contract. Counter-argument This is not a problem if the method uses a base exception. New exceptions can be thrown at will if the method declares throwing the base exception. The only requirement is that the new exceptions thrown are subclasses of the base exception.  Again, what is the value of having all methods that may throw exceptions declare throwing the same base exception? Does it enable you to handle the exceptions any better than if you knew the methods might throw an unchecked exception? Summary I used to be in favor of checked exceptions but recently I have begun to change my mind. Personalities like Rod Johnson (Spring Framework), Anders Hejlsberg (father of C#), Joshua Bloch (Effective Java, item 41: Avoid unnecessary use of checked exceptions) and others have made me rethink the real benefit of checked exceptions. Lately we have tried using unchecked exceptions on a larger project, and they have worked out just fine. The error handling is centralized in a few classes. Here and there we have had to do local error handling instead of propagating the exception to the main error handling code. But it is not in very many places. Our code has become somewhat more readable now that there aren't try-catch blocks all over the code. In other words, there are a lot less no-benefit catch-rethrow try-catch blocks in the code than with checked exceptions. All in all I would recommend using unchecked exceptions. At least give it a try on a project. I have summarized the reasons below:   * Unchecked exceptions do not clutter the code with unnecessary try-catch blocks. * Unchecked exceptions do not clutter the method declarations with aggregated exception declarations. * The argument that you easier forget to handle unchecked exceptions is not valid in my experience. * The argument that it is harder to know how to handle undeclared exceptions is not valid in my experience. * Unchecked exceptions avoids versioning problems altogether.   You or your project will have to make your own decisions about whether to use checked or unchecked exceptions, or both. Here is a list of resources that also discusses the decision between checked and unchecked exceptions.  Anders Hejlsberg on checked vs. unchecked exceptions [**http://www.artima.com/intv/handcuffs.html**](http://www.artima.com/intv/handcuffs.html)   James Gosling on checked exceptions [**http://www.artima.com/intv/solid.html**](http://www.artima.com/intv/solid.html)   Bill Venners on Exceptions [**http://www.artima.com/interfacedesign/exceptions.html**](http://www.artima.com/interfacedesign/exceptions.html)   Bruce Eckel on checked exceptions [**http://www.artima.com/intv/typingP.html**](http://www.artima.com/intv/typingP.html)   Designing with Exceptions (Bill Venners - www.artima.com) [**http://www.artima.com/designtechniques/desexcept.html**](http://www.artima.com/designtechniques/desexcept.html)   [**Effective Java**](http://www.amazon.co.uk/exec/obidos/redirect?link_code=ur2&camp=1634&tag=jenkovdevelop-21&creative=6738&path=ASIN/0201310058/qid=1124382514/sr=1-1/ref=sr_1_3_1)http://www.assoc-amazon.co.uk/e/ir?t=jenkovdevelop-21&l=ur2&o=2 (Joshua Bloch - Addison Wesley 2001)   Daniel Pietraru - in favor of checked exceptions [**Exceptional Java - Checked exceptions are priceless? For everything else there is the RuntimeException**](http://littletutorials.com/2008/05/06/exceptional-java-checked-exceptions-are-priceless-for-everything-else-there-is-the-the-runtimeexception/) | | |  | | --- | | **0. Chained exceptions**    :    [**Link**](http://docs.oracle.com/javase/tutorial/essential/exceptions/chained.html) |  Chained Exceptions An application often responds to an exception by throwing another exception. In effect, the first exception *causes* the second exception. It can be very helpful to know when one exception causes another. *Chained Exceptions* help the programmer do this.  The following are the methods and constructors in Throwable that support chained exceptions.  Throwable getCause()  Throwable initCause(Throwable)  Throwable(String, Throwable)  Throwable(Throwable)  The Throwable argument to initCause and the Throwable constructors is the exception that caused the current exception. getCause returns the exception that caused the current exception, and initCause sets the current exception's cause.  The following example shows how to use a chained exception.  try {  } catch (IOException e) {  throw new SampleException("Other IOException", e);  }  In this example, when an IOException is caught, a new SampleException exception is created with the original cause attached and the chain of exceptions is thrown up to the next higher level exception handler. Accessing Stack Trace Information Now let's suppose that the higher-level exception handler wants to dump the stack trace in its own format.  **Definition:** A stack trace provides information on the execution history of the current thread and lists the names of the classes and methods that were called at the point when the exception occurred. A stack trace is a useful debugging tool that you'll normally take advantage of when an exception has been thrown.  The following code shows how to call the getStackTrace method on the exception object.  catch (Exception cause) {  StackTraceElement elements[] = cause.getStackTrace();  for (int i = 0, n = elements.length; i < n; i++) {  System.err.println(elements[i].getFileName()  + ":" + elements[i].getLineNumber()  + ">> "  + elements[i].getMethodName() + "()");  }  } Logging API The next code snippet logs where an exception occurred from within the catch block. However, rather than manually parsing the stack trace and sending the output toSystem.err(), it sends the output to a file using the logging facility in the [java.util.logging](https://docs.oracle.com/javase/8/docs/api/java/util/logging/package-summary.html) package.  try {  Handler handler = new FileHandler("OutFile.log");  Logger.getLogger("").addHandler(handler);    } catch (IOException e) {  Logger logger = Logger.getLogger("package.name");  StackTraceElement elements[] = e.getStackTrace();  for (int i = 0, n = elements.length; i < n; i++) {  logger.log(Level.WARNING, elements[i].getMethodName());  }  } | | |  | | --- | | **0. Handling hierarchy of exceptions**    :    [**Link**](http://www.tutorialspoint.com/java/java_exceptions.htm) | | | |  | | --- | | **0. ‘finally’ clause**    :    [**Link**](http://www.javatpoint.com/finally-block-in-exception-handling) | | | |  | | --- | | **0. Assertions**    :    [**Link**](http://javarevisited.blogspot.in/2012/01/what-is-assertion-in-java-java.html) | | | |  | | --- | | **0. Custom Exceptions**    :    [**Link**](http://www.javatpoint.com/custom-exception) | | |  |

An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons, including the following:

* A user has entered invalid data.
* A file that needs to be opened cannot be found.
* A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

To understand how exception handling works in Java, you need to understand the three categories of exceptions:

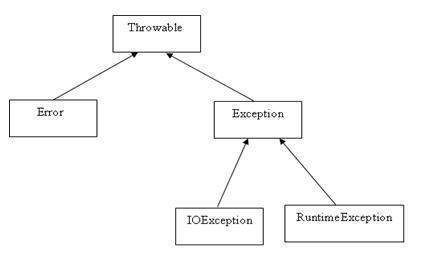
* **Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.
* **Runtime exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.
* **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Exception Hierarchy:

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are not normally trapped form the Java programs. These conditions normally happen in case of severe failures, which are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment. Example : JVM is out of Memory. Normally programs cannot recover from errors.

The Exception class has two main subclasses: IOException class and RuntimeException Class.



Here is a list of most common checked and unchecked [**Java's Built-in Exceptions**](http://www.tutorialspoint.com/java/java_builtin_exceptions.htm).

Exceptions Methods:

Following is the list of important medthods available in the Throwable class.

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **public String getMessage()**  Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor. |
| 2 | **public Throwable getCause()**  Returns the cause of the exception as represented by a Throwable object. |
| 3 | **public String toString()**  Returns the name of the class concatenated with the result of getMessage() |
| 4 | **public void printStackTrace()**  Prints the result of toString() along with the stack trace to System.err, the error output stream. |
| 5 | **public StackTraceElement [] getStackTrace()**  Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack. |
| 6 | **public Throwable fillInStackTrace()**  Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace. |

Catching Exceptions:

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

try

{

//Protected code

}catch(ExceptionName e1)

{

//Catch block

}

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example:

The following is an array is declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

// File Name : ExcepTest.java

import java.io.\*;

public class ExcepTest{

public static void main(String args[]){

try{

int a[] = new int[2];

System.out.println("Access element three :" + a[3]);

}catch(ArrayIndexOutOfBoundsException e){

System.out.println("Exception thrown :" + e);

}

System.out.println("Out of the block");

}

}

This would produce the following result:

Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3

Out of the block

Multiple catch Blocks:

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

try

{

//Protected code

}catch(ExceptionType1 e1)

{

//Catch block

}catch(ExceptionType2 e2)

{

//Catch block

}catch(ExceptionType3 e3)

{

//Catch block

}

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Example:

Here is code segment showing how to use multiple try/catch statements.

try

{

file = new FileInputStream(fileName);

x = (byte) file.read();

}catch(IOException i)

{

i.printStackTrace();

return -1;

}catch(FileNotFoundException f) //Not valid!

{

f.printStackTrace();

return -1;

}

The throws/throw Keywords:

If a method does not handle a checked exception, the method must declare it using the**throws**keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword. Try to understand the different in throws and throw keywords.

The following method declares that it throws a RemoteException:

import java.io.\*;

public class className

{

public void deposit(double amount) throws RemoteException

{

// Method implementation

throw new RemoteException();

}

//Remainder of class definition

}

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException:

import java.io.\*;

public class className

{

public void withdraw(double amount) throws RemoteException,

InsufficientFundsException

{

// Method implementation

}

//Remainder of class definition

}

The finally Keyword

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax:

try

{

//Protected code

}catch(ExceptionType1 e1)

{

//Catch block

}catch(ExceptionType2 e2)

{

//Catch block

}catch(ExceptionType3 e3)

{

//Catch block

}finally

{

//The finally block always executes.

}

Example:

public class ExcepTest{

public static void main(String args[]){

int a[] = new int[2];

try{

System.out.println("Access element three :" + a[3]);

}catch(ArrayIndexOutOfBoundsException e){

System.out.println("Exception thrown :" + e);

}

finally{

a[0] = 6;

System.out.println("First element value: " +a[0]);

System.out.println("The finally statement is executed");

}

}

}

This would produce the following result:

Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3

First element value: 6

The finally statement is executed

Note the following:

* A catch clause cannot exist without a try statement.
* It is not compulsory to have finally clauses when ever a try/catch block is present.
* The try block cannot be present without either catch clause or finally clause.
* Any code cannot be present in between the try, catch, finally blocks.

Declaring you own Exception:

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:

* All exceptions must be a child of Throwable.
* If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
* If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below:

class MyException extends Exception{

}

You just need to extend the Exception class to create your own Exception class. These are considered to be checked exceptions. The following InsufficientFundsException class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.

Example:

// File Name InsufficientFundsException.java

import java.io.\*;

public class InsufficientFundsException extends Exception

{

private double amount;

public InsufficientFundsException(double amount)

{

this.amount = amount;

}

public double getAmount()

{

return amount;

}

}

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

// File Name CheckingAccount.java

import java.io.\*;

public class CheckingAccount

{

private double balance;

private int number;

public CheckingAccount(int number)

{

this.number = number;

}

public void deposit(double amount)

{

balance += amount;

}

public void withdraw(double amount) throws

InsufficientFundsException

{

if(amount <= balance)

{

balance -= amount;

}

else

{

double needs = amount - balance;

throw new InsufficientFundsException(needs);

}

}

public double getBalance()

{

return balance;

}

public int getNumber()

{

return number;

}

}

The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

// File Name BankDemo.java

public class BankDemo

{

public static void main(String [] args)

{

CheckingAccount c = new CheckingAccount(101);

System.out.println("Depositing $500...");

c.deposit(500.00);

try

{

System.out.println("\nWithdrawing $100...");

c.withdraw(100.00);

System.out.println("\nWithdrawing $600...");

c.withdraw(600.00);

}catch(InsufficientFundsException e)

{

System.out.println("Sorry, but you are short $"

+ e.getAmount());

e.printStackTrace();

}

}

}

Compile all the above three files and run BankDemo, this would produce the following result:

Depositing $500...

Withdrawing $100...

Withdrawing $600...

Sorry, but you are short $200.0

InsufficientFundsException

at CheckingAccount.withdraw(CheckingAccount.java:25)

at BankDemo.main(BankDemo.java:13)

Common Exceptions:

In Java, it is possible to define two catergories of Exceptions and Errors.

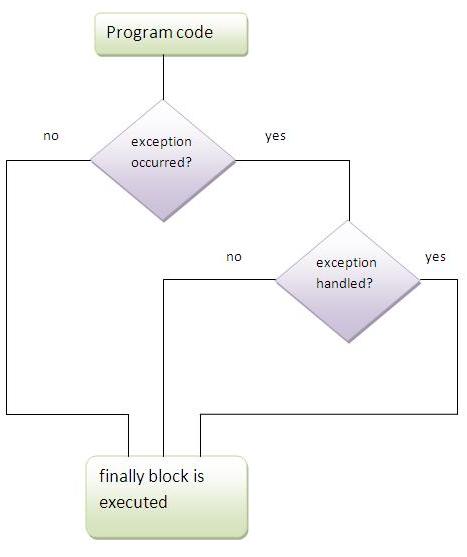
* **JVM Exceptions:** - These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples : NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException,
* **Programmatic exceptions:** - These exceptions are thrown explicitly by the application or the API programmers Examples: IllegalArgumentException, IllegalStateException.

# Java finally block

**Java finally block** is a block that is used to execute important code such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block must be followed by try or catch block.



#### Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).

## Why use java finally

* Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

## Usage of Java finally

Let's see the different cases where java finally block can be used.

### Case 1

Let's see the java finally example where **exception doesn't occur**.

1. **class** TestFinallyBlock{
2. **public** **static** **void** main(String args[]){
3. **try**{
4. **int** data=25/5;
5. System.out.println(data);
6. }
7. **catch**(NullPointerException e){System.out.println(e);}
8. **finally**{System.out.println("finally block is always executed");}
9. System.out.println("rest of the code...");
10. }
11. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestFinallyBlock)

Output:5

finally block is always executed

rest of the code...

### Case 2

Let's see the java finally example where **exception occurs and not handled**.

1. **class** TestFinallyBlock1{
2. **public** **static** **void** main(String args[]){
3. **try**{
4. **int** data=25/0;
5. System.out.println(data);
6. }
7. **catch**(NullPointerException e){System.out.println(e);}
8. **finally**{System.out.println("finally block is always executed");}
9. System.out.println("rest of the code...");
10. }
11. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestFinallyBlock1)

Output:finally block is always executed

Exception in thread main java.lang.ArithmeticException:/ by zero

### Case 3

Let's see the java finally example where **exception occurs and handled**.

1. **public** **class** TestFinallyBlock2{
2. **public** **static** **void** main(String args[]){
3. **try**{
4. **int** data=25/0;
5. System.out.println(data);
6. }
7. **catch**(ArithmeticException e){System.out.println(e);}
8. **finally**{System.out.println("finally block is always executed");}
9. System.out.println("rest of the code...");
10. }
11. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestFinallyBlock2)

Output:Exception in thread main java.lang.ArithmeticException:/ by zero

finally block is always executed

rest of the code...

#### Rule: For each try block there can be zero or more catch blocks, but only one finally block.

#### Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

# [What is Assertion in Java - Java Assertion Tutorial](http://javarevisited.blogspot.in/2012/01/what-is-assertion-in-java-java.html)

**Java Assertion** or **assert keyword in java** is little unknown and not many programmer is familiar with this and it's been rarely used specially if you have not writing unit test using JUnit which extensively uses Java assertion to compare test result. Junit itself is the biggest manifestation of what assertion in Java can do and believe me  by using assertion along with Exception you can write robust code. Assertion not only improve stability of code but also help you to become better programmer by forcing you to think about different scenario while [writing production quality code](http://javarevisited.blogspot.com/2011/09/how-to-write-production-quality-code.html) and improving your think through ability.

This article is in continuation of my earlier post like [How SubString works in Java](http://javarevisited.blogspot.com/2011/10/how-substring-in-java-works.html) or [Why main is public static in java](http://javarevisited.blogspot.com/2011/12/main-public-static-java-void-method-why.html), if you haven’t read  you may find interesting.

## Java Assertion tutorial - How Assertion works in Java

### What is assert keyword in Java

assert keyword is used to implement assertion in java. we can use assert keyword in two format

assert booleanExpression;

assert booleanExpression : errorMessage;

(here errorMessage can not be an invocation to a method with return type void)

### How Assertion works in Java

[What is assertion in Java, assert keyword in Java](http://javarevisited.blogspot.com/2011/12/bad-version-number-in-class-files-cause.html)As shown above assert keyword in java has two form first form "assert booleanExpression " is used to test the boolean expression and if boolean expression is false then java throws **AssertionError** and your program terminates. you can **use assert here to validate input or assumption** for example for a method which calculates stock price for trading , name of stock should not be null or empty, but as java recommends we should not use assertion to check arguments of public method instead public method should always check its argument and throw appropriate exception e.g. **IllegalArgumentException**.

Second form of assert keyword "assert booleanExpression : errorMessage" is more useful and provides a mechanism to pass additional data when Java assertion fails and java throws **AssertionError**.

### Benefit of using Assertion in Java

Assertion in Java offers several benefits to programmer if it used properly. even many of seasoned programmer has recommended using **assertion in Java** as good programming practice and good to add this point on your [code review checklist](http://javarevisited.blogspot.com/2011/09/code-review-checklist-best-practice.html) ? let's figure out *why using assert keyword is desired* and what benefit assertion offers :

1) assertion is simply great for **data validation**. In my opinion there is no better way then using java assert keyword to validate data passed to method, classic example is a method which calculates interest rates here we know that amount, time can not be less than zero by using assert keyword in java we can validate this data at run-time and if by any chance your function is getting incorrect data your program will fail with AssertionError.

2) Assertion in Java guarantees that at a certain point on function your **assumption** or certain condition is true otherwise it would not have come to that point and would have been terminated with AssertionError, this makes [debugging in Java](http://javarevisited.blogspot.com/2011/07/java-debugging-tutorial-example-tips.html) lot easier.

3) Using Java assert keyword helps to detect bug early in development cycle which is very **cost effective**. Assertion in Java also makes debugging easier to me AssertionError is a best kind of error while debugging because its clearly tell source and reason of error. Also code written using assert keyword fails close to source of error and require less time to find out the cause as compared to code **without assert keyword**.

4) assert statement in java is similar to **unit test** directly integrated into your code and have more chances to test your code with real word data than Junit test case, so it always **complement your Junit tests** or integration test suite.

5) writing code with assert statement will help you to be better programmer and **improve quality of code**, yes this is true based on my experience when we write code using assert statement we think through hard, we think about possible input to a function, we think about boundary condition which eventually result in better discipline and quality code.

6) assertion in java gives you lot of **confident while maintaining or refactoring code**, you can create new code and use assertion to compare output with old method if your program works well then you can simply comment out your old method.

// original code snippet:

int stockPrice = calculateStockPrice();

// code while refactoring and testing:

**assert(calculateStockPrice() == newCalculateStockPrice());**

int stockPrice = newCalculateStockPrice();

// code after refactoring completed:

int stockPrice = newCalculateStockPrice();

### Important point about Java assertion

1) Assertion is introduced in [JDK](http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html) 1.4 and implemented using *assert keyword in java*.

2) assertion can be enable and disable at run time by using switch -da or -disableassertion

3) Always remember Assertion does not replace Exception but compliments it.

4) Another important point is Assertion also doesn't replace need of unit testing instead if you see JUnit it shows how assertion can be useful to validate conditions.

5) do not use assertion to validate arguments or parameters of public method.

6) you can compile java code where assert is a legal identifier instead of keyword because by passing -source 1.3 during compilation. if you are working in java version 1.2 you can compile your code with assertion as below "

javac -source 1.4 OnlineStockTradingDemo.java

That’s all on Java Assertion, benefits of **assertion in Java** and where to use assertion in Java. Key point is Assertion should be thought as replacement of unit testing or Exception rather it compliments both of them , with **Java assertion** you can have more real world data for testing than unit testing.

Read more: <http://javarevisited.blogspot.com/2012/01/what-is-assertion-in-java-java.html#ixzz3gPjKGrZW>

**Chapter 16:Object Lifetime**

In Java,Garbage collection happens automatically during the lifetime of a java program, eliminating the need to de-allocate memory and avoiding memory leaks. In this tutorial we will learn the concept of GC in Java

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  | | --- | --- | | |  | | --- | | **0. Garbage Collection Tutorial : Memory allocation for an object Garbage Collection Garbage collection strategies Use of finalize()**    :    [**Link**](http://www.javatpoint.com/Garbage-Collection)  [**Link**](http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html) | | |  |

# Java Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

### Advantage of Garbage Collection

* It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
* It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

## How can an object be unreferenced?

There are many ways:

* By nulling the reference
* By assigning a reference to another
* By annonymous object etc.

### 1) By nulling a reference:

1. Employee e=**new** Employee();
2. e=**null**;

### 2) By assigning a reference to another:

1. Employee e1=**new** Employee();
2. Employee e2=**new** Employee();
3. e1=e2;//now the first object referred by e1 is available for garbage collection

### 3) By annonymous object:

1. **new** Employee();

## finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

1. **protected** **void** finalize(){}

#### Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

## gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

1. **public** **static** **void** gc(){}

#### Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

### Simple Example of garbage collection in java

1. **public** **class** TestGarbage1{
2. **public** **void** finalize(){System.out.println("object is garbage collected");}
3. **public** **static** **void** main(String args[]){
4. TestGarbage1 s1=**new** TestGarbage1();
5. TestGarbage1 s2=**new** TestGarbage1();
6. s1=**null**;
7. s2=**null**;
8. System.gc();
9. }
10. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestGarbage1)

object is garbage collected

object is garbage collected

#### Note: Neither finalization nor garbage collection is guaranteed.

**Chapter 17:Multithreading**

Java provides built-in support for multithreaded programming.Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.In this tutorial we will see how to create user threads, life cycle of a thread,thread synchronization, interthread communication and deadlocks scenerios.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | |  | | --- | | **0. Define Threads**    :    [**Link**](http://www.javabeginner.com/learn-java/java-threads-tutorial) |   **INTRODUCTION TO THREADS**  Multithreading refers to two or more tasks executing concurrently within a single program. A thread is an independent path of execution within a program. Many threads can run concurrently within a program. Every thread in Java is created and controlled by the **java.lang.Thread class**. A Java program can have many threads, and these threads can run concurrently, either asynchronously or synchronously.  Multithreading has several advantages over Multiprocessing such as;   * Threads are lightweight compared to processes * Threads share the same address space and therefore can share both data and code * Context switching between threads is usually less expensive than between processes * Cost of thread intercommunication is relatively low that that of process intercommunication * Threads allow different tasks to be performed concurrently.   The following figure shows the methods that are members of the Object and Thread Class.  http://www.javabeginner.com/images/table.JPG  **THREAD CREATION**  There are two ways to create thread in java;   * Implement the Runnable interface (java.lang.Runnable) * By Extending the Thread class (java.lang.Thread)   **IMPLEMENTING THE RUNNABLE INTERFACE**  **The Runnable Interface Signature**  public interface Runnable {  void run();  One way to create a thread in java is to implement the Runnable Interface and then instantiate an object of the class. We need to override the run() method into our class which is the only method that needs to be implemented. The run() method contains the logic of the thread.  **The procedure for creating threads based on the Runnable interface is as follows:**  1. A class implements the Runnable interface, providing the run() method that will be executed by the thread. An object of this class is a Runnable object.  2. An object of Thread class is created by passing a Runnable object as argument to the Thread constructor. The Thread object now has a Runnable object that implements the run() method.  3. The start() method is invoked on the Thread object created in the previous step. The start() method returns immediately after a thread has been spawned.  4. The thread ends when the run() method ends, either by normal completion or by throwing an uncaught exception.  Below is a program that illustrates instantiation and running of threads using the runnable interface instead of extending the Thread class. To start the thread you need to invoke the **start()** method on your object.  class RunnableThread implements Runnable {  Thread runner;  public RunnableThread() {  }  public RunnableThread(String threadName) {  runner = new Thread(this, threadName); // (1) Create a new thread.  System.out.println(runner.getName());  runner.start(); // (2) Start the thread.  }  public void run() {  //Display info about this particular thread  System.out.println(Thread.currentThread());  }  }  public class RunnableExample {  public static void main(String[] args) {  Thread thread1 = new Thread(new RunnableThread(), "thread1");  Thread thread2 = new Thread(new RunnableThread(), "thread2");  RunnableThread thread3 = new RunnableThread("thread3");  //Start the threads  thread1.start();  thread2.start();  try {  //delay for one second  Thread.currentThread().sleep(1000);  } catch (InterruptedException e) {  }  //Display info about the main thread  System.out.println(Thread.currentThread());  }  }  **Output**  thread3 Thread[thread1,5,main] Thread[thread2,5,main] Thread[thread3,5,main] Thread[main,5,main]private  [**Download**](http://www.wideskills.com/sites/default/files/Zip_Files/runnable-thread.zip)Runnable Thread Program Example  This approach of creating a thread by implementing the Runnable Interface must be used whenever the class being used to instantiate the thread object is required to extend some other class.  **EXTENDING THREAD CLASS**  The procedure for creating threads based on extending the Thread is as follows:  1. A class extending the Thread class overrides the run() method from the Thread class to define the code executed by the thread.  2. This subclass may call a Thread constructor explicitly in its constructors to initialize the thread, using the super() call.  3. The start() method inherited from the Thread class is invoked on the object of the class to make the thread eligible for running.  Below is a program that illustrates instantiation and running of threads by extending the Thread class instead of implementing the Runnable interface. To start the thread you need to invoke the **start()**method on your object.  class XThread extends Thread {  XThread() {  }  XThread(String threadName) {  super(threadName); // Initialize thread.  System.out.println(this);  start();  }  public void run() {  //Display info about this particular thread  System.out.println(Thread.currentThread().getName());  }  }  public class ThreadExample {  public static void main(String[] args) {  Thread thread1 = new Thread(new XThread(), "thread1");  Thread thread2 = new Thread(new XThread(), "thread2");  // The below 2 threads are assigned default names  Thread thread3 = new XThread();  Thread thread4 = new XThread();  Thread thread5 = new XThread("thread5");  //Start the threads  thread1.start();  thread2.start();  thread3.start();  thread4.starttry {  //The sleep() method is invoked on the main thread to cause a one second delay.  Thread.currentThread().sleep(1000);  } catch (InterruptedException e) {  }  //Display info about the main thread  System.out.println(Thread.currentThread());  }  }  **Output**  Thread[thread5,5,main] thread1 thread5 thread2 Thread-3 Thread-2 Thread[main,5,main]  [**Download**](http://www.wideskills.com/sites/default/files/Zip_Files/java-thread-example.zip)**Java Thread Program Example**  When creating threads, there are two reasons why implementing the Runnable interface may be preferable to extending the Thread class:   * Extending the Thread class means that the subclass cannot extend any other class, whereas a class implementing the Runnable interface has this option. * A class might only be interested in being runnable, and therefore, inheriting the full overhead of the Thread class would be excessive.   An example of an anonymous class below shows how to create a thread and start it:  ( new Thread() {  public void run() {  for(;;) System.out.println(“Stop the world!”);  }  }  ).start(); | | |  | | --- | | **0. Differentiate between Process and Thread**    :    [**Link**](http://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html) |  Processes and Threads In concurrent programming, there are two basic units of execution: *processes* and *threads*. In the Java programming language, concurrent programming is mostly concerned with threads. However, processes are also important.  A computer system normally has many active processes and threads. This is true even in systems that only have a single execution core, and thus only have one thread actually executing at any given moment. Processing time for a single core is shared among processes and threads through an OS feature called time slicing.  It's becoming more and more common for computer systems to have multiple processors or processors with multiple execution cores. This greatly enhances a system's capacity for concurrent execution of processes and threads — but concurrency is possible even on simple systems, without multiple processors or execution cores. Processes A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.  Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support *Inter Process Communication* (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes on the same system, but processes on different systems.  Most implementations of the Java virtual machine run as a single process. A Java application can create additional processes using a [ProcessBuilder](https://docs.oracle.com/javase/8/docs/api/java/lang/ProcessBuilder.html) object. Multiprocess applications are beyond the scope of this lesson. Threads Threads are sometimes called *lightweight processes*. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.  Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.  Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling. But from the application programmer's point of view, you start with just one thread, called the *main thread*. This thread has the ability to create additional threads, as we'll demonstrate in the next section. | | |  | | --- | | **0. Types of Threads**    :    [**Link**](http://www.xyzws.com/javafaq/what-is-difference-between-user-and-daemon-thread-in-java/196)  [**Link**](http://www.cs.umd.edu/~als/cmsc330-f99/projects/assign7.html) |  What is difference between User and Daemon Thread in Java? Java makes a distinction between a user thread and another type of thread known as a daemon thread. The daemon threads are typically used to perform services for user threads. The main() method of the application thread is a user thread. Threads created by a user thread are user thread. JVM doesn't terminates unless all the user thread terminate.  You can explicitly specify a thread created by a user thread to be a daemon thread by calling  setDaemon(true) on a Thread object. For example, the clock handler thread, the idle thread, the garbage collector thread, the screen updater thread, and the garbage collector thread are all daemon threads. A new created thread inherits the "daemon-status" of the thread that created it unless you explicitly calling setDaemon on that Thread object to change its status.  Note that the setDaemon() method must be called before the thread's start() method is invoked.Once a thread has started executing (i.e., its start() method has been called) its daemon status cannot be changed. To determine if a thread is a daemon thread, use the accessor method isDaemon().  The difference between these two types of threads is straightforward: If the Java runtime determines that the only threads running in an application are daemon threads (i.e., there are no user threads in existence) the Java runtime promptly closes down the application, effectively stopping all daemon threads dead in their tracks. In order for an application to continue running, it must always have at least one live user thread. In all other respects the Java runtime treats daemon threads and user threads in exactly the same manner. | | |  | | --- | | **0. Implement threads in java**    :    [**Link**](http://www.tutorialspoint.com/java/java_multithreading.htm)  [**Link**](http://etutorials.org/Misc/programmers+guide+java+certification/Chapter+9.+Threads/Programming+Exercises/) |   Multi-threading                          DUE: December 13, 1999 -- 6:00pm  In this assignment, you will use Java threads to implement players for a simple card game. The game has 4 players and 4 piles of cards, one between each pair of players. The players draw cards from the top and discard cards to the bottom of the piles in the following pattern:   |  |  |  | | --- | --- | --- | |  | Pile | | | Player | Draw | Discard | | 1 | 1 | 2 | | 2 | 2 | 3 | | 3 | 3 | 4 | | 4 | 4 | 1 |   The first player to collect 4 cards of the same denomination wins the game. The card deck has 32 cards: 4 of each denomination from 0 through 7.  However, your program must accept denominations greater than 7, for grading purposes (for testing and grading, we will create input decks with 4 of only one denomination between 0 and 7, 3 of the other denominations between 0 and 7, and fill out the 32 card deck with denominations greater than 7, with fewer than 4 of any of those denominations).   Initially each player is dealt 4 cards in round robin fashion starting from player 1, and the remainder of the cards are added to the piles, starting from Pile 1, in round robin fashion (with the first card added to each pile the first one drawn, as in a queue). Strategy So that the same player (usually) wins for the same input deck for all of you, everyone must use the same game playing strategy. The simple strategy you should use has each player preferring certain card denominations. In particular, player 1 prefers 0's and 1's, player 2 prefers 2's and 3's, player 3 prefers 4's and 5's, and player 4 prefers 6's and 7's. This means that (after drawing a card from the proper deck) a player discards any card from a non-preferred denomination, if possible. If that is not possible, then the player discards a card from the denomination with fewer cards. To ensure progress in the game, a player may not indefinitely hold a card from a non-preferred denomination.  This means that you must implement some mechanism for selecting a non-preferred denomination card that ensures that any particular non-preferred card is eventually selected for discard.  For example, if player 2 has (after drawing a card from the proper deck) one 2, one 4, one 5, and a pair of 3's, then the player would discard either the 4 or the 5. If player 3 had the same set of cards, the player would discard a 2 or a 3. An example of the other case, is that if player 2 had three 2's and a pair of 3's, then the player must discard a 3. Implementation In addition to a Main class that starts the game application, reads the input deck, deals the initial hands, and creates the various threads needed to run the game, you will need to build classes for the players and for the card piles. For the various classes, some of your methods will need to be synchronized, as discussed in recitation, to prevent multiple threads from accessing shared data simultaneously.  In addition, you will need to implement a routine that handles stopping players who did not win the game. The winning player can simply exit the run procedure that started its thread execution. However, this player must notify the other player threads that they need to stop (e.g., by setting a member value in the player class via a synchronized method, that is subsequently checked by all player threads). Each thread must print a message when it finishes (e.g., "Player 1 wins and exits", "Player 2 exits", etc.). Also, drawing and discarding is an atomic action, so a player must discard after a draw, even when the player wins.  This means a player always has 4 cards in the hand whenever printing the contents of a hand.  Each player must demonstrate card playing actions by printing the card drawn, the card discarded and the current hand for each action. The action must be labeled with the player number. Each player's actions should be written to a separate output file.  For example, intermediate moves should print something like the following:  **player 1 draws a 4 player 1 discards 4  player 1 hand 5 5 5 6** The final move of the winner prints something like the following: **player 3 draws a 7 player 3 discards a 3 player 3 hand 7 7 7 7 player 3 wins player 3 exits**  The card deck is read from standard input.  An example of an initial deck looks like this:  2 3 4 0 7 1 6 5 6 5 0 2 1 7 4 3 3 6 1 5 4 2 0 7 1 2 3 0 5 6 7 4  As the final output for each player at the end of the game, the player should print three lines, appropriately customized for that player, to the player's output file.  The first states whether the player won or lost, the second lists the player's hand at the end of the game, and the third lists the contents of the pile the player draws from at the end of the game.  The lines look like this, with required keywords in bold:  **WIN** yes (or no) **HAND** card1 card2 card3 card4 **DRAWPILE** contents of draw pile, separated by spaces (e.g., 2 5 3) Instructions for submitting your work  1. Put one class in each source file, and name the file with the class name (e.g., my\_class.java). 2. tar the Java file(s) and a Makefile for submission (e.g., **tar cvf submit7.tar \*.java**); 3. Your Java program should start from a class named Main (which must have a member function named main), read test data from an input data file specified on the command line, redirected from stdin, and write its output to four output files, which are always named player1.actions, player2.actions, player3.actions and player4.actions . 4. Submit the tar file: **~as330003/alpha.bin/submit 7 submit7.tar**.   Your work may not be graded if these procedures are not followed exactly. | | |  | | --- | | **0. Life cycle of a thread and thread states**    :    [**Link**](http://www.roseindia.net/java/thread/life-cycle-of-threads.shtml) |   Java is a*multithreaded programming language* which means we can develop multithreaded program using Java. A multithreaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.  By definition multitasking is when multiple processes share common processing resources such as a CPU. Multithreading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.  Multithreading enables you to write in a way where multiple activities can proceed concurrently in the same program. Life Cycle of a Thread: A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.  Java Thread  Above-mentioned stages are explained here:   * **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread. * **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task. * **Waiting:** Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task.A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing. * **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs. * **Terminated:**A runnable thread enters the terminated state when it completes its task or otherwise terminates.  Thread Priorities: Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.  Java thread priorities are in the range between MIN\_PRIORITY (a constant of 1) and MAX\_PRIORITY (a constant of 10). By default, every thread is given priority NORM\_PRIORITY (a constant of 5).  Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependentant. Create Thread by Implementing Runnable Interface: If your class is intended to be executed as a thread then you can achieve this by implementing **Runnable** interface. You will need to follow three basic steps: Step 1: As a first step you need to implement a run() method provided by **Runnable** interface. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of run() method:  public void run( ) Step 2: At second step you will instantiate a **Thread** object using the following constructor:  Thread(Runnable threadObj, String threadName);  Where, *threadObj* is an instance of a class that implements the **Runnable** interface and**threadName** is the name given to the new thread. Step 3 Once Thread object is created, you can start it by calling **start( )** method, which executes a call to run( ) method. Following is simple syntax of start() method:  void start( ); Example: Here is an example that creates a new thread and starts it running:  class RunnableDemo implements Runnable {  private Thread t;  private String threadName;    RunnableDemo( String name){  threadName = name;  System.out.println("Creating " + threadName );  }  public void run() {  System.out.println("Running " + threadName );  try {  for(int i = 4; i > 0; i--) {  System.out.println("Thread: " + threadName + ", " + i);  // Let the thread sleep for a while.  Thread.sleep(50);  }  } catch (InterruptedException e) {  System.out.println("Thread " + threadName + " interrupted.");  }  System.out.println("Thread " + threadName + " exiting.");  }    public void start ()  {  System.out.println("Starting " + threadName );  if (t == null)  {  t = new Thread (this, threadName);  t.start ();  }  }  }  public class TestThread {  public static void main(String args[]) {    RunnableDemo R1 = new RunnableDemo( "Thread-1");  R1.start();    RunnableDemo R2 = new RunnableDemo( "Thread-2");  R2.start();  }  }  This would produce the following result:  Creating Thread-1  Starting Thread-1  Creating Thread-2  Starting Thread-2  Running Thread-1  Thread: Thread-1, 4  Running Thread-2  Thread: Thread-2, 4  Thread: Thread-1, 3  Thread: Thread-2, 3  Thread: Thread-1, 2  Thread: Thread-2, 2  Thread: Thread-1, 1  Thread: Thread-2, 1  Thread Thread-1 exiting.  Thread Thread-2 exiting. Create Thread by Extending Thread Class: The second way to create a thread is to create a new class that extends **Thread** class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class. Step 1 You will need to override **run( )** method available in Thread class. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of run() method:  public void run( ) Step 2 Once Thread object is created, you can start it by calling **start( )** method, which executes a call to run( ) method. Following is simple syntax of start() method:  void start( ); Example: Here is the preceding program rewritten to extend Thread:  class ThreadDemo extends Thread {  private Thread t;  private String threadName;    ThreadDemo( String name){  threadName = name;  System.out.println("Creating " + threadName );  }  public void run() {  System.out.println("Running " + threadName );  try {  for(int i = 4; i > 0; i--) {  System.out.println("Thread: " + threadName + ", " + i);  // Let the thread sleep for a while.  Thread.sleep(50);  }  } catch (InterruptedException e) {  System.out.println("Thread " + threadName + " interrupted.");  }  System.out.println("Thread " + threadName + " exiting.");  }    public void start ()  {  System.out.println("Starting " + threadName );  if (t == null)  {  t = new Thread (this, threadName);  t.start ();  }  }  }  public class TestThread {  public static void main(String args[]) {    ThreadDemo T1 = new ThreadDemo( "Thread-1");  T1.start();    ThreadDemo T2 = new ThreadDemo( "Thread-2");  T2.start();  }  }  This would produce the following result:  Creating Thread-1  Starting Thread-1  Creating Thread-2  Starting Thread-2  Running Thread-1  Thread: Thread-1, 4  Running Thread-2  Thread: Thread-2, 4  Thread: Thread-1, 3  Thread: Thread-2, 3  Thread: Thread-1, 2  Thread: Thread-2, 2  Thread: Thread-1, 1  Thread: Thread-2, 1  Thread Thread-1 exiting.  Thread Thread-2 exiting. Thread Methods: Following is the list of important methods available in the Thread class.   |  |  | | --- | --- | | **SN** | **Methods with Description** | | 1 | **public void start()**  Starts the thread in a separate path of execution, then invokes the run() method on this Thread object. | | 2 | **public void run()**  If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable  object. | | 3 | **public final void setName(String name)**  Changes the name of the Thread object. There is also a getName() method for retrieving the name. | | 4 | **public final void setPriority(int priority)**  Sets the priority of this Thread object. The possible values are between 1 and 10. | | 5 | **public final void setDaemon(boolean on)**  A parameter of true denotes this Thread as a daemon thread. | | 6 | **public final void join(long millisec)**  The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes. | | 7 | **public void interrupt()**  Interrupts this thread, causing it to continue execution if it was blocked for any reason. | | 8 | **public final boolean isAlive()**  Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion. |   The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread.   |  |  | | --- | --- | | **SN** | **Methods with Description** | | 1 | **public static void yield()**  Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled. | | 2 | **public static void sleep(long millisec)**  Causes the currently running thread to block for at least the specified number of milliseconds. | | 3 | **public static boolean holdsLock(Object x)**  Returns true if the current thread holds the lock on the given Object. | | 4 | **public static Thread currentThread()**  Returns a reference to the currently running thread, which is the thread that invokes this method. | | 5 | **public static void dumpStack()**  Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application. |  Example: The following ThreadClassDemo program demonstrates some of these methods of the Thread class. Consider a class **DisplayMessage** which implements **Runnable**:  // File Name : DisplayMessage.java  // Create a thread to implement Runnable  public class DisplayMessage implements Runnable  {  private String message;  public DisplayMessage(String message)  {  this.message = message;  }  public void run()  {  while(true)  {  System.out.println(message);  }  }  }  Following is another class which extends Thread class:  // File Name : GuessANumber.java  // Create a thread to extentd Thread  public class GuessANumber extends Thread  {  private int number;  public GuessANumber(int number)  {  this.number = number;  }  public void run()  {  int counter = 0;  int guess = 0;  do  {  guess = (int) (Math.random() \* 100 + 1);  System.out.println(this.getName()  + " guesses " + guess);  counter++;  }while(guess != number);  System.out.println("\*\* Correct! " + this.getName()  + " in " + counter + " guesses.\*\*");  }  }  Following is the main program which makes use of above defined classes:  // File Name : ThreadClassDemo.java  public class ThreadClassDemo  {  public static void main(String [] args)  {  Runnable hello = new DisplayMessage("Hello");  Thread thread1 = new Thread(hello);  thread1.setDaemon(true);  thread1.setName("hello");  System.out.println("Starting hello thread...");  thread1.start();    Runnable bye = new DisplayMessage("Goodbye");  Thread thread2 = new Thread(bye);  thread2.setPriority(Thread.MIN\_PRIORITY);  thread2.setDaemon(true);  System.out.println("Starting goodbye thread...");  thread2.start();  System.out.println("Starting thread3...");  Thread thread3 = new GuessANumber(27);  thread3.start();  try  {  thread3.join();  }catch(InterruptedException e)  {  System.out.println("Thread interrupted.");  }  System.out.println("Starting thread4...");  Thread thread4 = new GuessANumber(75);    thread4.start();  System.out.println("main() is ending...");  }  }  This would produce the following result. You can try this example again and again and you would get different result every time.  Starting hello thread...  Starting goodbye thread...  Hello  Hello  Hello  Hello  Hello  Hello  Goodbye  Goodbye  Goodbye  Goodbye  Goodbye  ....... Major Java Multithreading Concepts: While doing Multithreading programming in Java, you would need to have the following concepts very handy:   * [What is thread synchronization?](http://www.tutorialspoint.com/java/java_thread_synchronization.htm) * [Handling threads inter communication](http://www.tutorialspoint.com/java/java_thread_communication.htm) * [Handling thread deadlock](http://www.tutorialspoint.com/java/java_thread_deadlock.htm) * [Major thread operations](http://www.tutorialspoint.com/java/java_thread_control.htm)  Life Cycle of A Thread **[http://www.roseindia.net/images/previous.gif](http://www.roseindia.net/java/thread/overview-of-thread.shtml)**    **[http://www.roseindia.net/images/bt_home.gif](http://www.roseindia.net/java/thread/index.shtml)**  **[http://www.roseindia.net/images/next.gif](http://www.roseindia.net/java/thread/thread-creation.shtml)**  When you are programming with threads, understanding the life cycle of thread is very valuable. While a thread is alive, it is in one of several states. By invoking start() method, it doesn?t mean that the thread has access to CPU and start executing straight away. Several factors determine how it will proceed.  **Different states of a thread are :**  http://www.roseindia.net/java/thread/life-c1.gif  Advertisement   1. **New state ?** After the creations of Thread instance the thread is in this state but before the start() method invocation. At this point, the thread is considered not alive. 2. **Runnable (Ready-to-run) state ?**A thread start its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor. 3. **Running state ?**A thread is in running state that means the thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in Running state: the scheduler select a thread from runnable pool. 4. **Dead state ?** A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again. 5. **Blocked -**A thread can enter in this state because of waiting the resources that are hold by another thread.   **Different states implementing Multiple-Threads are:**  http://www.roseindia.net/java/thread/multi-thread/JavaMu3.gif  As we have seen different states that may be occur with the single thread. A running thread can enter to any non-runnable state, depending on the circumstances. A thread cannot enters directly to the running state from non-runnable state, firstly it goes to runnable state. Now lets understand the some non-runnable states which may be occur handling the multithreads.   * **Sleeping ?** On this state, the thread is still alive but it is not runnable, it might be return to runnable state later, if a particular event occurs. On this state a thread sleeps for a specified amount of time. You can use the method **sleep( )** to stop the running state of a thread.       **static void sleep(long millisecond) throws InterruptedException** * **Waiting for Notification ?**A thread waits for notification from another thread. The thread sends back to runnable state after sending notification from another thread.     **final void wait(long timeout) throws InterruptedException    final void wait(long timeout, int nanos) throws InterruptedException    final void wait() throws InterruptedException** * **Blocked on I/O ?** The thread waits for completion of blocking operation. A thread can enter on this state because of waiting I/O resource. In that case the thread sends back to runnable state after availability of resources. * **Blocked for joint completion ?** The thread can come on this state because of waiting the completion of another thread. * **Blocked for lock acquisition ?** The thread can come on this state because of waiting to acquire the lock of an object.   **Methods that can be applied apply on a Thread:**  Some Important Methods defined in**java.lang.Thread**are shown in the table:   |  |  |  | | --- | --- | --- | | **Method** | **Return Type** | **Description** | | currentThread( ) | Thread | Returns an object reference to the thread in which it is invoked. | | getName( ) | String | Retrieve the name of the thread object or instance. | | start( ) | void | Start the thread by calling its runmethod. | | run( ) | void | This method is the entry point to execute thread, like the main method for applications. | | sleep( ) | void | Suspends a thread for a specified amount of time (in milliseconds). | | isAlive( ) | boolean | This method is used to determine the thread is running or not. | | activeCount( ) | int | This method returns the number of active threads in a particular thread group and all its subgroups. | | interrupt( ) | void | The method interrupt the threads on which it is invoked. | | yield( ) | void | By invoking this method the current thread pause its execution temporarily and allow other threads to execute. | | join( ) | void | This method and **join(long millisec)** Throws InterruptedException.  These two methods are invoked on a thread. These are not returned until either the thread has completed or it is timed out respectively. | | | |  | | --- | | **0. Important methods in Multithreading**    :    [**Link**](http://www.javajee.com/java-multithreading-methods) |   Thread class has many important methods like init, start, stop, run, getName(), getPriority(), isAlive() and join(), and also few static methods like sleep(), yield() etc. Usually we override only the run() method and use the inherited versions of init, start etc   ****start() and run()**** Invoking the**start()** method on a thread object executes the **run()** method as a new thread of execution. When you call the start method for a thread object, it will call a native code method that causes the OS to initiate another thread from which the run() method executes. We can call start method only once on a thread or it will throw IllegalStateException.  You can also call the run method directly, but if you call the run() method directly, it would simply operate like any other method and will run as part of the same thread that called it.   final ****isAlive()**** The final **isAlive()** method returns true if the thread is still running or the Thread has not terminated.   final ****join()**** The final **join()** method waits until thread on which it is called is terminated. For example, thread1.join() suspends the current thread until thread1 dies.  You can also pass a long value to join method to specify the number of milliseconds you are prepared to wait for the death of a thread. For example, thread1.join(1000) wait upto 1 second for thread1 to die, and then continue execution.  The join() method can throw an InterruptedException if the current thread is interrupted by another thread.   ****yield()**** Calling **yield()** will move the current thread from running to runnable, to give other threads a chance to execute. However the scheduler may still bring the same thread back to running. A better alternative according to Effective Java by Joshua Bloch is to use Thread.sleep(1) instead of Thread.yield().   ****Wait, notify and notifyAll**** Java.lang.Object provides three methods – notify(), notifyAll() and wait () – to improve the efficiency communication between threads. You will need to understand the [synchronization process in Java](http://javajee.com/basics-of-synchronization-in-java) to understand the communication using wait, notify and notifyAll.   * **obj.wait()** makes a thread wait on an object (obj) until it receives a notification from a notify() or notifyAll() on the same object. * **obj.notify()** sends a notification to any one waiting thread on ab object (obj) that the object lock  is available. * **obj.notifyAll()** sends notification to all waiting threads on an object (obj) that the object lock is available.   You should call the above methods inside a synchronized block, or you will get IllegalMonitorStateException. The IllegalMonitorStateException is thrown to indicate that a thread has attempted to wait on an object's monitor or to notify other threads waiting on an object's monitor without owning the specified monitor.  The wait, notify and notifyAll methods should be used with caution for thread communication; if not used properly, it may result in [deadlock](http://javajee.com/deadlock-in-java-with-example).   ****interrupt(), isInterrupted() and interrupted()**** Thread’s stop() method is deprecated and should not be used. Instead, you should use the interrupt mechanism.  A thread can signal another thread that it should stop executing by calling the **interrupt() method** for that thread. This doesn’t stop the thread, but sets a flag in the thread. This flag must be checked in the run() method to have any effect and the thread should then terminate itself.  The**isInterrupted() method**returns true if the interrupted flag has been set. This method does not reset the flag, but another **static method interrupted()**tests the flag for the currently executing thread and if it has been interrupted, it clears the interrupted flag in the current thread object and returns true.  When an **InterruptedException** is thrown, the flag that registers the interrupt is cleared, so a subsequent call to isInterrupted() or interrupted() returns false.   ****static currentThread()**** The static **currentThread()**method returns a reference to the thread in which it is called.   ****static sleep()**** This static **sleep()**method causes the thread to suspend execution for a given time. The sleep method has two overloaded versions:   * static void sleep (long milliseconds) throws InterruptedException * static void sleep (long milliseconds, int nanoseconds) throws InterruptedException    ****Sleep vs Wait**** Both sleep() and wait() methods can also be used to suspend a thread of execution for a specified time. The difference is that, when sleep() is executed inside a synchronized block, the object still holds the lock, but when wait() method is executed, it releases the lock and breaks the synchronization block, so other threads can acquire the lock.  The wait() method is used in connection with notify() or notifyAll() methods in thread communication. A waiting thread comes out of waiting when some other thread notify it using notify() or notifyAll() on the same object on which it is waiting. There is an overloaded version of the wait that takes a time as input and comes out of wait when that time is over even though no one has notified using notify() or notifyAll().   ****Deprecated Methods**** Thread.**stop**, Thread.**suspend**and Thread.**resume**are deprecated. You can find the reasons for deprecating them at: | | |  | | --- | | **0. Thread priorities and thread scheduling**    :    [**Link**](http://swapdhalange.blogspot.in/2012/02/explain-thread-priority-in-java-with.html) |   If you are aware of interprocess communication then it will be easy for you to understand inter thread communication. Inter thread communication is important when you develop an application where two or more threads exchange some information.  There are simply three methods and a little trick which makes thread communication possible. First let's see all the three methods listed below:   |  |  | | --- | --- | | **SN** | **Methods with Description** | | 1 | **public void wait()**  Causes the current thread to wait until another thread invokes the notify(). | | 2 | **public void notify()**  Wakes up a single thread that is waiting on this object's monitor. | | 3 | **public void notifyAll()**  Wakes up all the threads that called wait( ) on the same object. |   These methods have been implemented as **final** methods in Object, so they are available in all the classes. All three methods can be called only from within a **synchronized** context.  Example:  This examples shows how two thread can communicate using **wait()** and **notify()** method. You can create a complex system using the same concept.  class Chat {  boolean flag = false;  public synchronized void Question(String msg) {  if (flag) {  try {  wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  System.out.println(msg);  flag = true;  notify();  }  public synchronized void Answer(String msg) {  if (!flag) {  try {  wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  System.out.println(msg);  flag = false;  notify();  }  }  class T1 implements Runnable {  Chat m;  String[] s1 = { "Hi", "How are you ?", "I am also doing fine!" };  public T1(Chat m1) {  this.m = m1;  new Thread(this, "Question").start();  }  public void run() {  for (int i = 0; i < s1.length; i++) {  m.Question(s1[i]);  }  }  }  class T2 implements Runnable {  Chat m;  String[] s2 = { "Hi", "I am good, what about you?", "Great!" };  public T2(Chat m2) {  this.m = m2;  new Thread(this, "Answer").start();  }  public void run() {  for (int i = 0; i < s2.length; i++) {  m.Answer(s2[i]);  }  }  }  public class TestThread {  public static void main(String[] args) {  Chat m = new Chat();  new T1(m);  new T2(m);  }  }  When above program is complied and executed, it produces following result:  Hi  Hi  How are you ?  I am good, what about you?  I am also doing fine!  Great! | | |  | | --- | | **0. Thread synchronization and inter-threaded communication**    :    [**Link**](http://www.tutorialspoint.com/java/java_thread_communication.htm) | | |