What is java?

Classpath

Class path is the folder that contains .class files and java language related data.

1. Temporary class path-using cmd promt
2. Permanent path- modifying system environmental variable

Variable

Variable is the name of memory area or memeory reference,( where we are going to store our data) each memory area store one specific data type. Its value may vary at runtime.

1. Instance
2. Static
3. local

Operators

Operators are symbols they can perform a specific kind of task by following their precedence order.

Keyword

Java keywords are 50 reserved words and here each key word has a specific meaning.

So we can’t use these key words for other purposes

Ex:- short, strictfp,class etc…

Datatype

Data types used to define two things first one is range or size of data and format of the data.

1. primitive datatypes (char, Boolean, long) (it is set of data types that holds fixed size of data )
2. non-primitive (holds the reference of object) (it is set of datatypes that holds variable data types)

Wrapper class

Wrapper classes are java classes , they exist in java.lang package.

They are used to convert primitive data types to objects. Here each class can handle one datatype.

Control statements

It is sequential execution block , it will execute until given condition is satisfyied.

Statement

Statement is specific action in java program . it can perform one particular task.

1. Declaration
2. Expression
3. Control flow

String

String is a java class . it can store sequence of characters as string literals or string object.

Create class

Immutable class

The class which can resist modification once it created is called immutable class.

To make one class as immutable we should make that class as

1. Don’t provide setters
2. Make fields as private
3. Don’t allow subclasses to modify methods.

Pass by value

System properties

System.getProperty() to retrieve all system properties

Static

Static is a keyword. We can use this keyword on variable, method, import statement,static block, class(inner class). If we make anything as static then they can be call with class name.

Static import

Importing the static members of classes is called static import. It comes with two varities

1. Single static import
2. Static import on demand

Hashcode() vs Equals()

hashCode() method is used to get hashcode and Equals() method is used to check Quality of object.

Incase of object comparison hashcode checks that whether the two objects contains same hash code or not . and comes to Equals method it will checks that the two objects contains same data or not.

This and super

This is a keyword. It holds reference of current instance of class. Generally this is used to call current class data member , method and constructor. And this also used to separate data members and local variables in ambiguity situation.

Super is a keyword. It holds the reference of parent class object. Generally Super keyword is used in inheritance concept to call parent class data member, method and constructor.

32bit vs 64bit architecture

32 bit processor can handle the data upto 4GB but 64 Bit can handle the data upto 196 GB

Java.exe vs javaw.exe

Java.exe is console version , this version is used to interact java with text based iputs.

Javaw.exe is non-console version , this version is used interact java with GUI.

Commandline args

The arguments passed at the time at launching the java application is called commandline args. Generally this method is used in console applications.

Compare floats

Comparing float values ,do not give true even they are equal becoz of precesion value diff. to over come this problem 3 aproaches are there

1. Simple comparison-(using ==)
2. Thereshold based comparioson—(comparing with third variable)

Final float thereshold= 0.0001;

If((val1-val2)<thershold) {}

1. Compare with BigDesimal(adding values and comparing values with BigDecimal Object methods)

Recursion

Recursion means method calls itself until given condition is satisfied.

Pair class

Javafx.util.Pair is simplest way to create key, value

Pair<integer,String> obj= new pair<>(123, “bhaskar”);

Obj.getKey();

Obj.getValue();

ImmutablePair

Same as above but its object is immutable

Tuple2 class

Tubple

(j7) diamond oparartor

While creating generics object , specifying parametres on right side is optional from java 7this feature is diamond operator.

(j7) string in switch

Before java 7 only enum, int , character is used in switch after java introduces String also.

(j7) try with resources

From java 7 onwards relinquishing the resource in try block is optional, this method is called try with resource.

Legacy

Try{

Resource will be created

}

Finally{resource will be relinquished}

New

Try( resource will be created) {

}

In try with resource creating finally block is optional

(j7) number formatting

From java 7 onwards \_ symbol is used separate numeric values for easy reading purpose

Ex:- 1\_00\_00\_000

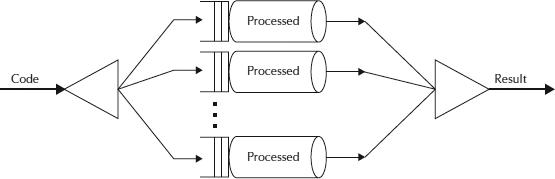
(j7) suppressed exceptions

(j7) multiple exception in single cache block

Cache(NullPointerExeption | ArithematicExeption ex) {}

(j7) forkjoin

Basically the **Fork-Join breaks the task at hand into mini-tasks** until the mini-task is simple enough that it can be solved without further breakups. It’s **like a** [**divide-and-conquer algorithm**](https://en.wikipedia.org/wiki/Divide_and_conquer_algorithms). One important concept to note in this framework is that **ideally no worker thread is idle**. They implement a **work-stealing algorithm** in that idle workers steal the work from those workers who are busy.



It’s based on the work of Doug Lea, a thought leader on Java concurrency. Fork/Join deals with the threading hassles; you just indicate to the framework which portions of the work can be broken apart and handled recursively.

Refer examples

(j7) forkjoin vs executors

(j7) WatchService (auto reload properties) (api)

It is interface ,it acts like as listener class for configuration file.it will automatically detects the changes and reload the changes.

(j8) forEach

Java forEach is utility method it can iterate over collection and streams and makes it available to it(collections and streams)

|  |
| --- |
| Iterable.java |
| default void forEach(Consumer<? super T> action) {      Objects.requireNonNull(action);      for (T t : this) {          action.accept(t);      }  } |

(j8) stream

Stream is sequence of data flow coming from source. here source may be collection or arrays.

Refer how to do in java

(j8) boxed stream

To use primitive types in streams first we have to convert them into its wrapper classes(boxing) , this process is called boxed streams.

(j8) lamda expressions

Lambda expression is an anonymous function that contains implementation without method name and giving parametres is optional .

Refer How to do in java

(j8) functional interface

The interface contains only one abstract method is called functional interface.

These interfaces are marked with annotation @FunctionalInterface (allows only one abstract method). We can implement it with lamda expressions.

(j8) method references

It is lambda expression concept , using this concept we can refer methods with their class

Math::max equivalent to Math.max(x,y) (static)

System.out::println equivalent to System.out.println(x) (instance)

ArrayList::new equivalent to new ArrayList()

(j8) default method

Fully implemented method declared in interface is called default method.

Default method is used in a situation where we need call multiple same methods from different interfaces.

Walkable.super.move();

//or

Moveable.super.move();

(j8) Optional class

Optional is one of the new class created in java8. It is mainly introduced to resolve NullPointer Exeption problem . becoz we cant create it with null reference.

See documentation

(j8) Predicate

Predicate in general meaning is a statement about something that is either true or false. In programming, predicates represent single argument functions that return a boolean value.

com/zetcode/JavaPredicateEx.java

package com.zetcode;

import java.util.List;

import java.util.function.Predicate;

class BiggerThanFive<E> implements Predicate<Integer> {

@Override

public boolean test(Integer v) {

Integer five = 5;

return v > five;

}

}

public class JavaPredicateEx {

public static void main(String[] args) {

List<Integer> nums = List.of(2, 3, 1, 5, 6, 7, 8, 9, 12);

BiggerThanFive<Integer> btf = new BiggerThanFive<>();

nums.stream().filter(btf).forEach(System.out::println);

}

}

(j8) Date time

Java provides the **Date** class available in **java.util** package, The class **Date** represents a specific instant in **time**, with millisecond precision. ... **Date**(long milliseconds) : Creates a **date** object for the given milliseconds.

old date api do not fulfil all the requirement so they introduced a new date classes

New classes to represent local date and timezone

New classes to represent timestamp and duration

Added utility classes over existing enums

Date adjusters introduced

Building dates will be easier

New class to simulate system/machine clock

Timezone handling related changes

Date formatting changes

#### LocalTime

The [LocalTime](https://docs.oracle.com/javase/8/docs/api/java/time/LocalTime.html) class represents a time. There is no representation of a date or time-zone.

|  |
| --- |
| //LocalTime localTime = LocalTime.now();     //toString() in format 09:57:59.744  LocalTime localTime = LocalTime.of(12, 20);  System.out.println(localTime.toString());    //12:20  System.out.println(localTime.getHour());     //12  System.out.println(localTime.getMinute());   //20  System.out.println(localTime.getSecond());   //0  System.out.println(localTime.MIDNIGHT);      //00:00  System.out.println(localTime.NOON);          //12:00 |

#### LocalDateTime

The [LocalDateTime](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html) class represents a date-time. There is no representation of a time-zone.

|  |
| --- |
| LocalDateTime localDateTime = LocalDateTime.now();  System.out.println(localDateTime.toString());      //2013-05-15T10:01:14.911  System.out.println(localDateTime.getDayOfMonth()); //15  System.out.println(localDateTime.getHour());       //10  System.out.println(localDateTime.getNano());       //911000000 |

Instant

Instant instant = Instant.now();

System.out.println(instant.toString());                                 //2013-05-15T05:20:08.145Z

System.out.println(instant.plus(Duration.ofMillis(5000)).toString());   //2013-05-15T05:20:13.145Z

System.out.println(instant.minus(Duration.ofMillis(5000)).toString());  //2013-05-15T05:20:03.145Z

System.out.println(instant.minusSeconds(10).toString());                //2013-05-15T05:

#### Duration

[Duration](https://docs.oracle.com/javase/8/docs/api/java/time/Duration.html) class is a whole new concept brought first time in java language. It represents the time difference between two time stamps.

|  |
| --- |
| Duration duration = Duration.ofMillis(5000);  System.out.println(duration.toString());     //PT5S    duration = Duration.ofSeconds(60);  System.out.println(duration.toString());     //PT1M |

(j8) Iterate directory

(j8) Read file

(j8) write tp fiile

//Get the file reference

Path path = Paths.get("c:/output.txt");

//Use try-with-resource to get auto-closeable writer instance

try (BufferedWriter writer = Files.newBufferedWriter(path))

{

    writer.write("Hello World !!");

}

(j8) String to Date

String anotherDate = "04 Apr 2016";

DateTimeFormatter df = DateTimeFormatter.ofPattern("dd MMM yyyy");

LocalDate random = LocalDate.parse(anotherDate, df);

System.out.println(anotherDate + " parses as " + random);

(j8) joinArray

Java examples to **join string array** to produce single String. This code can be used to **convert array to string** in Java. We may need this information many times during development specially while parsing contents out of JSON or XML.

List<String> strList = Arrays.asList("How", "To", "Do", "In", "Java");

String joinedString = String.join(", ", strList);

System.out.println(joinedString);

Output:

How, To, Do, In, Java

(j8) Base64

Base 64 encoding convert your binary data into 64 printable ASCII characters. Generally it is done for binary data in email messages and "basic" HTTP authentication. These 64 printable characters are:

* 26 uppercase letters [A…Z]
* 26 lowercase letters [a…z]
* 10 digits [0…9]
* 2 symbols [[Read more](https://en.wikipedia.org/wiki/Base64#Implementations_and_history)]

|  |
| --- |
| * Base64.Encoder encoder = Base64.getEncoder(); * String normalString = "username:password"; * String encodedString = encoder.encodeToString( * normalString.getBytes(StandardCharsets.UTF\_8) ); * Output: * dXNlcm5hbWU6cGFzc3dvcmQ= |

#### 2) Decoding a base 64 encoded string

This is also very simple. Just get the instance of Base64.Decoder and use it to decode the base 64 encoded string.

|  |
| --- |
| String encodedString = "dXNlcm5hbWU6cGFzc3dvcmQ=";  Base64.Decoder decoder = Base64.getDecoder();  byte[] decodedByteArray = decoder.decode(encodedString);  //Verify the decoded string  System.out.println(new String(decodedByteArray));    Output:    username:password |

(j8) Comparartor

(j8) Regex vs Predicate

(j8) join String

String joined = String.join("/","usr","local","bin");

System.out.println(joined);

Output:

usr/local/bin

see o/p

(j8) diff. b/w Dates Internal and External

(j8) Secure Random

Oops

Oops is way of writing programming language with the concept of idea called “object”.

It mainly contains 4 principles

1. Abstraction
2. Encapsulation
3. Inheritance
4. polymorphism

Access modifiers

Access modifiers are java key words . which can apply restriction on class, method and data members like which can access it or which cant access it. Java has 4 types access modifiers

1. public
2. protected (same package and sub class)
3. deafault (same package)
4. private (same class)

Contructors

Constructors are Special methods in java class . its name same as class name and it should not have return type. It is invoked using new keyword. It perform following task

1. creates object of its own class
2. initializes the datamembers with its default value
3. executes the set of statements that is defined within the constructor

Instance Initializers

The block that is defined with curly braces within the class and outside the method is called instance block (instance initializers) . this block will execute just before constructor goes to execute.

Abstraction

Hiding implementation and showing only functionalities.

Ex:- methods

Encapsulation

Hiding data using access modifiers. We can achieve it by applying private access modifier on data members.

Ex:- private int a=3;

Inheritance (is-A)

The process of acquiring functionalities from its parent class is called inheritance.

We can create inheritance between two classes using extends

Types of inheritance

1. Single (1 to 1)
2. Multilevel (1 to 1 to 1)
3. Hierarchical (1 to 3)
4. Multiple (2 to 1) (not possible by classes) (possible through default methods)

Polymorphism

Polymorphism is the ability of functions and reference variable which behaves differently at different application context.

1. Static polymorphism

The flow of control is decided in compile time only. We can achieve it through method overloading.

1. Dynamic polymorphism

The flow of control is decided in runtime only. We can achive it through method overriding.

Overloading

Creating multiple methods with same method name and with the difference of signatures.

Rules : should have same return type, and different signatures (type , order, number of parametres), must throw same exeptions

overriding

redefining the parent class method in child class.

Rules:

1. Both methods should have same signatures 2.
2. return type may same type or its covariant type
3. Private,static, final methods and variables cannot be overridden.
4. Overridden method cannot throw checked exeption hier in hierarchiy.
5. Overridden methods cannot reduce its access scope

Interface

Interfaces is a set of rules and guidelines . and this rules and guidelines provided in the form of public static final data members and public abstract methods. These will guide the its implementation classes like what kind of methods it should implement and what type or results it should return

Abstract class (partial behavior)

The class which is defined with abstract key word is called abstract class. This class may or may not contains abstract (un defined methods) methods .

Extends vs implements

instanceOf operator

instanceOf is type checking operator . it checks the particular object is instance of the specific type or not (class or subclass or interface). It always returns true or false.

Multiple inheritance

Association

Objects have an independent life cycle.

Ex:- Student-teacher

Weather forecast app with proxy servers

Aggretation

Objects have independent life cycle but there is owner ship.

Ex:- cell phone, battery

Weather forecast with no proxy servers

Composition

Don’t have an independant life cycle.

Ex:- Question-answer

Irctc app and website

Concurrency

Concurrency is the ability to run different types of programs or different parts of a program parallel. in java concurrency is achieved through multi threading. Here cpu time slicing happens.

Why: to achieve high performance and minimize execution time.

Thread safety

Thread safety is a concept by implementing this concept we can share a data between multiple threads with accurate results.

Search for Example

Concurancy vs parallelism

Parallelism means multiple task executing parallel at same time. Execution order should fallow . Each task will be assigned to one core of a processor.

Compare and swap

Synchronized keyword

Synchronization means allowing sequential access flow. It means it will allow only one access at a time.

Synchronized is java access specifier. If we define anything with synchronized key word then it allow only one thread at a time.

Read Examples

Object vs Class level locking

Object level locking means applying synchronized key word on instance data members, methods and blocks.

3 ways to create it

public class DemoClass

{

    public synchronized void demoMethod(){}

}

or

public class DemoClass

{

    public void demoMethod(){

        synchronized (this)

        {

            //other thread safe code

        }

    }

}

or

public class DemoClass

{

    private final Object lock = new Object();

    public void demoMethod(){

        synchronized (lock)

        {

            //other thread safe code

        }

    }

}

Class level locking means applying synchronized key word on static method.

|  |
| --- |
| Various ways for class level locking |
| public class DemoClass  {      //Method is static      public synchronized static void demoMethod(){        }  }    or    public class DemoClass  {      public void demoMethod()      {          //Acquire lock on .class reference          synchronized (DemoClass.class)          {              //other thread safe code          }      }  }    or    public class DemoClass  {      private final static Object lock = new Object();        public void demoMethod()      {          //Lock object is static          synchronized (lock)          {              //other thread safe code          }      }  } |

Runnable vs Thread

Runnable is interface , it has one abstract method run(). If we want create a thread using Runnable then make one class implementing Runnable. To get the functionalities of thread create one thread object and pass runnable object as constructor param.

Thread is a java class. If we want to create a thread with Thread class then make that class extending from Thread and override run() method.

Wait(), notify(), notifyAll()

Wait is used to move one thread from running state to waiting state, notify is used to make move one thread from wait state to running state

Yield() vs join()

Yield() is used to apply low execution priority between group of same priority threads.

Join() is used make a thread to wait until the above threads completes its execution

Sleep() vs Wait()

Sleep will move thread to waiting state until certain time.

Wait method move thread to waiting state until notify method invoke it.

Lock vs Monitor

Callable vs Future

**1. Callable**

Callable interface has the call() method. In this method, we have to implement the logic of a task. The Callable interface is a parameterized interface, meaning we have to indicate the type of data the call() method will return.

**2. Future**

Future interface has methods to obtain the result generated by a Callable object and to manage its state.

UnCaughtExceptionHandler

in run() method we cannt make that class throwing checked exception using throws key word to resolve that issue useUnCaughtExceptionHandler.

class ExceptionHandler implements UncaughtExceptionHandler

{

   public void uncaughtException(Thread t, Throwable e)

   {

      System.out.printf("An exception has been captured\n");

      System.out.printf("Thread: %s\n", t.getId());

      System.out.printf("Exception: %s: %s\n", e.getClass().getName(), e.getMessage());

      System.out.printf("Stack Trace: \n");

      e.printStackTrace(System.out);

      System.out.printf("Thread status: %s\n", t.getState());

      new Thread(new Task()).start();

   }

}

Throttling Task Submission

Executor Best practises

InterThread communication

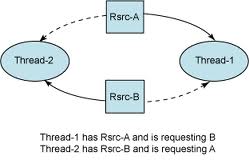
Write and resolve Deadlock

So, to solve it, we will simply re-order the statements where the code is accessing shared resources.

|  |
| --- |
| ResolveDeadLockTest.java |
| // Thread-1  Runnable block1 = new Runnable() {      public void run() {          synchronized (b) {              try {                  // Adding delay so that both threads can start trying to                  // lock resources                  Thread.sleep(100);              } catch (InterruptedException e) {                  e.printStackTrace();              }              // Thread-1 have A but need B also              synchronized (a) {                  System.out.println("In block 1");              }          }      }  };    // Thread-2  Runnable block2 = new Runnable() {      public void run() {          synchronized (b) {              // Thread-2 have B but need A also              synchronized (a) {                  System.out.println("In block 2");              }          }      }  }; |

Deadlock

In Java, a [deadlock](https://en.wikipedia.org/wiki/Deadlock) is a situation where minimum two threads are holding the lock on some different resource, and both are waiting for other’s resource to complete its task. And, none is able to leave the lock on the resource it is holding.



AtomicInteger

The **AtomicInteger** class protects an underlying int value by providing methods that perform **atomic operations** on the value. The AtomicInteger class is part of the java.util.concurrent.atomic package since Java

|  |
| --- |
| AtomicInteger example |
| //Initial value is 0  AtomicInteger atomicInteger = new AtomicInteger();    //Initial value is 100  AtomicInteger atomicInteger = new AtomicInteger(100);    int currentValue = atomicInteger.get();         //100    atomicInteger.set(1234);                        //Now value is 1234 |

Lock (interface)

A java.util.concurrent.locks.Lock is a thread synchronization mechanism just like synchronized blocks.

Since Lock is an interface, you need to use one of its implementations to use a Lock in your applications. ReentrantLock is one such implementation of Lock interface.

Lock lock = new ReentrantLock();

lock.lock();

//critical section

lock.unlock();

## Difference between Lock Interface and synchronized keyword

The main differences between a Lock and a synchronized block are:

1) Having a timeout trying to get access to a synchronized block is not possible. Using [Lock.tryLock(long timeout, TimeUnit timeUnit)](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html#tryLock%28long,%20java.util.concurrent.TimeUnit%29), it is possible.  
2) The synchronized block must be fully contained within a single method. A Lock can have it’s calls to lock() and unlock() in separate methods.

ThreadFactory (interface)

The [**factory design pattern**](file:///D:\websites\howtodoinjava.com\design-patterns\creational\implementing-factory-design-pattern-in-java\index.htm) is one of the most used design patterns in the java. It is one of [**creational patterns**](file:///\\howtodoinjava.com\category\design-patterns\creational\) and can be used to develop an object in demand of one or several classes. With this factory, we centralize the creation of objects.

The centralization of creation logic brings us some advantages e.g.

1. It’s easy to change the class of the objects created or the way we create these objects.
2. It’s easy to limit the creation of objects for limited resources. For example, we can only have N objects of a type.
3. It’s easy to generate statistical data about the creation of the objects.

We can create custom thread factory by implemting this.

ThreadLocal

The Java Concurrency API provides a clean mechanism for thread-local variables using [**ThreadLocal**](https://docs.oracle.com/javase/7/docs/api/java/lang/ThreadLocal.html) class with a very good performance.

|  |
| --- |
| public class ThreadLocal<T> extends Object {...} |

This class provides thread-local variables.

These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable. ThreadLocal instances are typically ***private static*** fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID).

This class has following methods:

1. **get()** : Returns the value in the current thread’s copy of this thread-local variable.
2. **initialValue()** : Returns the current thread’s “initial value” for this thread-local variable.
3. **remove()** : Removes the current thread’s value for this thread-local variable.
4. **set(T value)** : Sets the current thread’s copy of this thread-local variable to the specified value.

class DemoTask implements Runnable

{

   // Atomic integer containing the next thread ID to be assigned

   private static final AtomicInteger        nextId   = new AtomicInteger(0);

   // Thread local variable containing each thread's ID

   private static final ThreadLocal<Integer> threadId = new ThreadLocal<Integer>()

                                                         {

                                                            @Override

                                                            protected Integer initialValue()

                                                            {

                                                               return nextId.getAndIncrement();

                                                            }

                                                         };

   // Returns the current thread's unique ID, assigning it if necessary

   public int getThreadId()

   {

      return threadId.get();

   }

   // Returns the current thread's starting timestamp

   private static final ThreadLocal<Date> startDate = new ThreadLocal<Date>()

                                                 {

                                                    protected Date initialValue()

                                                    {

                                                       return new Date();

                                                    }

                                                 };

ExecutorService

In simple Java applications, we do not face much challenge while working with a small number of threads. If you have to develop a program that runs a lot of concurrent tasks, this approach will present many disadvantages such as lots of boiler plate code (create and manage threads), executing thread manually and keeping track of thread execution results.

The framework consist of three main interfaces (and lots of child interfaces) i.e. **Executor**, **ExecutorService** and [**ThreadPoolExecutor**](file:///D:\websites\howtodoinjava.com\java\multi-threading\java-thread-pool-executor-example\index.htm).

#### Benefits of Executor framework

* The framework mainly separates task creation and execution. Task creation is mainly boiler plate code and is easily replaceable.
* With an executor, we have to create tasks which implement either Runnable or Callable interface and send them to the executor.
* Executor internally maintain a (configurable) thread pool to improve application performance by avoiding the continuous spawning of threads.
* Executor is responsible for executing the tasks, running them with the necessary threads from the pool.

#### Execute Runnable tasks

We can execute runnables using the following methods :

* **void execute(Runnable task)** – executes the given command at some time in the future.
* **Future submit(Runnable task)** – submits a runnable task for execution and returns a Future representing that task. The Future’s get() method will return null upon successful completion.
* **Future submit(Runnable task, T result)** – Submits a runnable task for execution and returns a Future representing that task. The Future’s get() method will return the given result upon successful completion.

  //Executor service instance

        ExecutorService executor = Executors.newFixedThreadPool(10);

        //1. execute task using execute() method

        executor.execute(runnableTask);

        //2. execute task using submit() method

        Future<String> result = executor.submit(runnableTask, "DONE");

        //Shut down the executor service

        executor.shutdownNow();

        //Executor service instance

        ExecutorService executor = Executors.newFixedThreadPool(1);

        List<Callable<String>> tasksList = Arrays.asList(callableTask, callableTask, callableTask);

        //1. execute tasks list using invokeAll() method

        try

        {

            List<Future<String>> results = executor.invokeAll(tasksList);

            for(Future<String> result : results) {

                System.out.println(result.get());

            }

ThreadPoolExecutor

[Creating a thread in Java](https://howtodoinjava.com/java/multi-threading/difference-between-implements-runnable-and-extends-thread-in-java/) is an expensive operation. And if you start creating new thread instance everytime to execute a task, application performance will degrade surely.

A **thread pool is a collection of pre-initialized threads**. Generally the size of collection is fixed, but it is not mandatory. It facilitates the execution of N number of tasks using same threads. If thread are more tasks than threads, then tasks need to wait in a queue like structure ([FIFO – First in first out](https://en.wikipedia.org/wiki/FIFO_and_LIFO_accounting#FIFO)).

ThreadPoolExecutor separates the task creation and its execution. With ThreadPoolExecutor, you only have to implement the Runnable objects and send them to the executor. It is responsible for their execution, instantiation, and running with necessary threads.

We can create following 5 types of thread pool executors with pre-built methods in java.util.concurrent.Executors interface.

1. **Fixed thread pool executor** – Creates a thread pool that reuses a fixed number of threads to execute any number of tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. It is best fit for most off the real-life usecases.

|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(10); |

1. **Cached thread pool executor** – Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. DO NOT use this thread pool if tasks are long running. It can bring down the system if number of threads goes beyond what system can handle.

|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newCachedThreadPool(); |

1. **Scheduled thread pool executor** – Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newScheduledThreadPool(10); |

1. **Single thread pool executor** – Creates single thread to execute all tasks. Ute it when you have only one task to execute.

|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newSingleThreadExecutor(); |

1. **Work stealing thread pool executor** – Creates a thread pool that maintains enough threads to support the given parallelism level. Here parallelism level means the maximum number of threads which will be used to execute a given task, at single point of time, in multi-processor machines.

|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newWorkStealingPool(4); |

FixedSizeThreadPoolExecutor

SheduledThreadPoolExecutor

## Execute a task after a period of time

ScheduledExecutorService executor = Executors.newScheduledThreadPool(2);

Task task1 = new Task ("Demo Task 1");

Task task2 = new Task ("Demo Task 2");

System.out.println("The time is : " + new Date());

executor.schedule(task1, 5 , TimeUnit.SECONDS);

executor.schedule(task2, 10 , TimeUnit.SECONDS);

try {

executor.awaitTermination(1, TimeUnit.DAYS);

} catch (InterruptedException e) {

e.printStackTrace();

}

executor.shutdown();

}

}

Output:

The time is : Wed Mar 25 16:14:07 IST 2015

Doing a task during : Demo Task 1 - Time - Wed Mar 25 16:14:12 IST 2015

Doing a task during : Demo Task 2 - Time - Wed Mar 25 16:14:17 IST 2015

Semephore

A semaphore is a counter that protects the access to one or more shared resources.

when a thread wants to access one of shared resources (guarded by semaphore), first, it must acquire the semaphore. If the internal counter of the semaphore is greater than 0, the semaphore decrements the counter and allows access to the shared resource. Otherwise, if the counter of the semaphore is 0, the semaphore puts the thread to sleep until the counter is greater than 0. A value of 0 in the counter means all the shared resources are used by other threads, so the thread that wants to use one of them must wait until one is free.

When a thread has finished the use of the shared resource, it must release the semaphore so that the other threads can access the shared resource. That operation increases the internal counter of the semaphore.

 //This Semaphore will keep track of no. of printers used at any point of time.

    private final Semaphore semaphore;

    //While checking/acquiring a free printer out of three available printers, we will use this lock.

    private final Lock printerLock;

 //While checking/acquiring a free printer out of three available printers, we will use this lock.

    private final Lock printerLock;

    //This array represents the pool of free printers.

    private boolean freePrinters[];

    public PrinterQueue()

    {

        semaphore = new Semaphore(3);

        freePrinters = new boolean[3];

        Arrays.fill(freePrinters, true);

        printerLock = new ReentrantLock();

    }

    public void printJob(Object document)

    {

        try

        {

            //Decrease the semaphore counter to mark a printer busy

            semaphore.acquire();

            //Get the free printer

            int assignedPrinter = getPrinter();

            //Print the job

            Long duration = (long) (Math.random() \* 10000);

            System.out.println(Thread.currentThread().getName()

                    + ": Printer " + assignedPrinter

                    + " : Printing a Job during " + (duration / 1000)

                    + " seconds :: Time - " + new Date());

            Thread.sleep(duration);

            //Printing is done; Free the printer to be used by other threads.

            releasePrinter(assignedPrinter);

        }

        catch (InterruptedException e) {

            e.printStackTrace();

        }

        finally {

            System.out.printf("%s: The document has been printed\n", Thread

                    .currentThread().getName());

            //Increase the semaphore counter back

            semaphore.release();

        }

    }

    //Acquire a free printer for printing a job

    private int getPrinter()

    {

        int foundPrinter = -1;

        try {

            //Get a lock here so that only one thread can go beyond this at a time

            printerLock.lock();

            //Check which printer is free

            for (int i = 0; i < freePrinters.length; i++)

            {

                //If free printer found then mark it busy

                if (freePrinters[i])

                {

                    foundPrinter = i;

                    freePrinters[i] = false;

                    break;

                }

            }

        }

        catch (Exception e) {

            e.printStackTrace();

        } finally

        {

            //Allow other threads to check for free priniter

            printerLock.unlock();

        }

        return foundPrinter;

    }

    //Release the printer

    private void releasePrinter(int i) {

        printerLock.lock();

        //Mark the printer free

        freePrinters[i] = true;

        printerLock.unlock();

    }

}

BinarySemaphore

**binary semaphore can have a value either 0 or 1**. It means **binary semaphore protect the access to a SINGLE shared resource**, so the internal counter of the semaphore can only take the values 1 or 0.

So whenever you have a requirement for protecting the access to a SINGLE resource accessed by multiple threads, you can use Binary Semaphore.

BlockinQueue

DelayQueue

**DelayQueue** class is an unbounded blocking queue of delayed elements, in which an element can only be taken when its delay has expired. DelayQueue class is part of java.util.concurrent package.

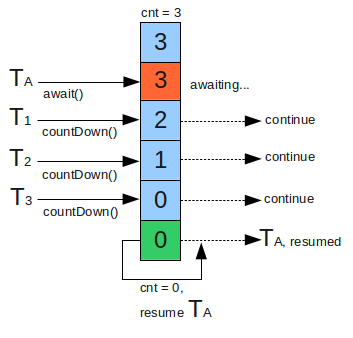
|  |
| --- |
| DelayedEvent.java |
| class DelayedEvent implements Delayed  {      private long id;      private String name;      private LocalDateTime activationDateTime;        public DelayedEvent(long id, String name, LocalDateTime activationDateTime) {          super();          this.id = id;          this.name = name;          this.activationDateTime = activationDateTime;      }        public long getId() {          return id;      }        public String getName() {          return name;      }        public LocalDateTime getActivationDateTime() {          return activationDateTime;      }        @Override      public int compareTo(Delayed that)      {          long result = this.getDelay(TimeUnit.NANOSECONDS)                          - that.getDelay(TimeUnit.NANOSECONDS);          if (result < 0) {              return -1;          } else if (result > 0) {              return 1;          }          return 0;      }        @Override      public long getDelay(TimeUnit unit) {          LocalDateTime now = LocalDateTime.now();          long diff = now.until(activationDateTime, ChronoUnit.MILLIS);          return unit.convert(diff, TimeUnit.MILLISECONDS);      }    } |

ConcurrentLinkedDeque

CountDownLatch

CountDownLatch was **introduced with JDK 1.5 along with other concurrent utilities like CyclicBarrier, Semaphore,** [**ConcurrentHashMap**](file:///\\howtodoinjava.com\java\collections\best-practices-for-using-concurrenthashmap\) **and** [**BlockingQueue**](file:///\\howtodoinjava.com\java-5\how-to-use-blockingqueue-and-threadpoolexecutor-in-java\) in java.util.concurrent package. This class **enables a java thread to wait until other set of threads completes** their tasks. e.g. Application’s main thread want to wait, till other service threads which are responsible for starting framework services have completed started all services.

CountDownLatch works by having a counter initialized with number of threads, which is decremented each time a thread complete its execution. When count reaches to zero, it means all threads have completed their execution, and thread waiting on latch resume the execution.

CountDownLatch Concept

Pseudo code for CountDownLatch can be written like this:

//Main thread start

//Create CountDownLatch for N threads

//Create and start N threads

//Main thread wait on latch

//N threads completes there tasks are returns

//Main thread resume execution

//Get latch object in constructor so that after completing the task, thread can countDown() the latch

    public BaseHealthChecker(String serviceName, CountDownLatch latch)

    {

        super();

        this.\_latch = latch;

        this.\_serviceName = serviceName;

        this.\_serviceUp = false;

    }

    @Override

    public void run() {

        try {

            verifyService();

            \_serviceUp = true;

        } catch (Throwable t) {

            t.printStackTrace(System.err);

            \_serviceUp = false;

        } finally {

            if(\_latch != null) {

                \_latch.countDown();

            }

        }

 //Initialize the latch with number of service checkers

        \_latch = new CountDownLatch(3);

        //All add checker in lists

        \_services = new ArrayList<BaseHealthChecker>();

        \_services.add(new NetworkHealthChecker(\_latch));

        \_services.add(new CacheHealthChecker(\_latch));

        \_services.add(new DatabaseHealthChecker(\_latch));

ForkJoinPool

Array:

An array is a collection of homogeneous data types. The length of an array is established when the array is created. After creation, its length is fixed.

🡪int[] intArray= new int[6]; created new array with their default values irrespective pf their scopes

🡪memory allocation : heap memory, continues memory allocation happens, fixed length and create memory for data not assigning areas, length is not extendable

🡪Extending array length : create second array with required length and copy it. Or System.arrayCopy(first array, first array length, second array, second array length,how many elemts to copy)

🡪var Args: public void add(int… numbers) , creating multiple methods with overloaded perametres are bad practise create var args instead of this.

1. If we call hundreds of times var args will create anonymous arrays it will effect the performance.
2. To resolve this add some overloaded methids with most commonly calling.

🡪Remove duplication : Assert.assertArrayEquals(newArray,removeDuplicationWithSort.removeDuplicates(oldArray))

ArrayList

arrayList is one of the class in java.util. package. It creates dynamic array to store data(variable length) . it works only with objects not primitives.

🡪contains static nature like arrays intital size 10 after exceeding its capacity creates new array of size= old arrayList size+ half of its old length , 10,15,22…

Solution:

List<Student> students= new ArrayList<Students>(20);

use when lot of reads occurs with less modification.

LinkedList

Linked list is one of the class in java.util package. It is used store the data in the form of nodes. Here each node contains two parts a) adrees part b) data part. Here address part contains the location of next node. Here it stores the data in insertion order.

* It allows only sequenciel access. Cannt retrieve specific position element as ArrayList and Vector.
* Null-> 22->23->43->21->null

If elements left side Is null it will treat it as first element and if pointing to right null it will treat as last element.use when lot of modification occurs with less reads.

HashMap (not thread safe)

Java HashMap is a [hash table](https://en.wikipedia.org/wiki/Hash_table) based implementation of Java’s Map interface. A Map, as you might know, is a collection of key-value pairs. It maps keys to values.

🡪types of retrieving map values

1. Use enhanced for loop to to get key values using getKey() use it to get value using getValue() method.
2. Use enhanced for loop to to get values value using getValue() method.
3. Use Map.Entry interface to get to get all keys and values using entrySet().

🡪it can allows null values as key and values. And doesn’t allow duplicate keys.

🡪it is not thread safe

🡪hash map internally implemented with an array called Bucket. If you are trying to put new values in HashMap first checks for Bucket array key value present or not using equals() method. If present return same reference otherwise creates new LinkedList object to store key, value.

🡪Hashmap default capacity is 16 if data reaches to 75% (Load factor) of full capacity then it will create new bucket array to hashmap with capacity 2x16 means doubled(32)

Hashtable

A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.

🡪doesn’t allow null values

🡪it is thread safe

Q: why String is mostly used as key in java

A: becoz it is immutable so no modification allowed& Integer object also immutable

Note: map collection always use hasCode() method to store and retrieve key’s. like pointing to string constant pool.

LinkedHashMap

**LinkedHashMap** is a Hash table and linked list implementation of the Map interface, with predictable iteration order. ... This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order).

🡪it will store insertion order as how we originally store into it. It means it maintains insertion order.

🡪it requires more memory than HashMap.

TreeMap (non synchronized)

treeMap is implements the interface NavigableMap and extends the AbstractMap. It can store the key , value pair in the form of sorted order on based on key.

Java TreeMap contains only unique elements.

Java TreeMap cannot have a null key but can have multiple null values.

Java TreeMap is non synchronized.

Java TreeMap maintains ascending order.

HashSet (non synchronized)

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

* HashSet stores the elements by using a mechanism called **hashing.**
* HashSet contains unique elements only.
* HashSet allows null value.
* HashSet class is non synchronized.
* HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
* HashSet is the best approach for search operations.
* The initial default capacity of HashSet is 16, and the load factor is 0.75.

LinkedHashSet

Java LinkedHashSet class is a Hashtable and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.

The important points about Java LinkedHashSet class are:

* Java LinkedHashSet class contains unique elements only like HashSet.
* Java LinkedHashSet class provides all optional set operation and permits null elements.
* Java LinkedHashSet class is non synchronized.
* Java LinkedHashSet class maintains insertion order.

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

* Java TreeSet class contains unique elements only like HashSet.
* Java TreeSet class access and retrieval times are quiet fast.
* Java TreeSet class doesn't allow null element.
* Java TreeSet class is non synchronized.
* Java TreeSet class maintains ascending order.

Comparable

Java Comparable interface is used to order the objects of the user-defined class. This interface is found in java.lang package and contains only one method named compareTo(Object). It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only. For example, it may be rollno, name, age or anything else.

### compareTo(Object obj) method

1. class Student implements Comparable<Student>{
2. int rollno;
3. String name;
4. int age;
5. Student(int rollno,String name,int age){
6. this.rollno=rollno;
7. this.name=name;
8. this.age=age;
9. }
11. public int compareTo(Student st){
12. if(age==st.age)
13. return 0;
14. else if(age>st.age)
15. return 1;
16. else
17. return -1;
18. }
19. }

Comparator

**Java Comparator interface** is used to order the objects of a user-defined class.

This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).

It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

#### Method of Collections class for sorting List elements

**public void sort(List list, Comparator c):** is used to sort the elements of List by the given Comparator.

Ex:-  import java.util.\*;

 class AgeComparator implements Comparator{

 public int compare(Object o1,Object o2){

 Student s1=(Student)o1;

 Student s2=(Student)o2;



 if(s1.age==s2.age)

 return 0;

 else if(s1.age>s2.age)

 return 1;

 else

 return -1;

 }

🡪main class

 Collections.sort(al,new NameComparator());

 Iterator itr=al.iterator();

 while(itr.hasNext()){

 Student st=(Student)itr.next();

 System.out.println(st.rollno+" "+st.name+" "+st.age);

 }

Iterator

Iterator is an interface it is used to retrieve all the elements from collection(list, set, map)

ListIterator

**Java ListIterator** interface is **bi-directional iterator** which is used to iterate over the elements of [list](file:///D:\websites\howtodoinjava.com\java-arraylist\index.htm) in either direction previous or next.

Spliterator

Java Spliterator interface is an internal iterator that breaks the [stream](https://howtodoinjava.com/java8/java-8-tutorial-streams-by-examples/) into the smaller parts. These smaller parts can be processed in parallel.

In real life programming, we may never need to use Spliterator directly. Under normal operations, it will behave exactly same as Java [Iterator](file:///D:\websites\howtodoinjava.com\java\collections\java-iterator\index.htm).

|  |
| --- |
| Spliterator Syntax |
| Spliterator<T> spliterator = list.spliterator(); |

PriorityQueue

|  |
| --- |
| **Java PriorityQueue** class is a queue data structure implementation in which objects are processed based on their **priority**.  PriorityQueue Example |
| //Comparator for name field  Comparator<Employee> nameSorter = Comparator.comparing(Employee::getName);    PriorityQueue<Employee> priorityQueue = new PriorityQueue<>( nameSorter );    priorityQueue.add(new Employee(1l, "AAA", LocalDate.now()));  priorityQueue.add(new Employee(4l, "CCC", LocalDate.now()));  priorityQueue.add(new Employee(5l, "BBB", LocalDate.now()));  priorityQueue.add(new Employee(2l, "FFF", LocalDate.now()));  priorityQueue.add(new Employee(3l, "DDD", LocalDate.now()));  priorityQueue.add(new Employee(6l, "EEE", LocalDate.now()));    while(true)  {      Employee e = priorityQueue.poll();      System.out.println(e);        if(e == null) break;  } |

priorityBlockingQueue

**Java PriorityBlockingQueue** class is **concurrent** blocking queue data structure implementation in which objects are processed based on their **priority**. The “blocking” part of the name is added to imply the **thread will block waiting until there’s an item available on the queue**.

|  |
| --- |
| PriorityBlockingQueue Example |
| //Comparator for name field  Comparator<Employee> nameSorter = Comparator.comparing(Employee::getName);    PriorityBlockingQueue<Employee> PriorityBlockingQueue = new PriorityBlockingQueue<>( 11, nameSorter );    PriorityBlockingQueue.add(new Employee(1l, "AAA", LocalDate.now()));  PriorityBlockingQueue.add(new Employee(4l, "CCC", LocalDate.now()));  PriorityBlockingQueue.add(new Employee(5l, "BBB", LocalDate.now()));  PriorityBlockingQueue.add(new Employee(2l, "FFF", LocalDate.now()));  PriorityBlockingQueue.add(new Employee(3l, "DDD", LocalDate.now()));  PriorityBlockingQueue.add(new Employee(6l, "EEE", LocalDate.now()));    while(true)  {      Employee e = PriorityBlockingQueue.poll();      System.out.println(e);        if(e == null) break;  } |

ArrayBlockinQueue

**ArrayBlockingQueue** class is Java **concurrent** and **bounded** blocking queue implementation backed by an array. It orders elements FIFO (first-in-first-out).

The **head** of the ArrayBlockingQueue is that element that has been on the queue the longest time. The **tail** of the ArrayBlockingQueue is that element that has been on the queue the shortest time. New **elements are inserted at the tail** of the queue, and the queue **retrieval operations obtain elements at the head** of the queue.

## 1. ArrayBlockingQueue Features

Let’s note down few important points on the ArrayBlockingQueue class.

* ArrayBlockingQueue is a bounded queue of fixed size backed by an array.
* It orders elements FIFO (first-in-first-out).
* Elements are inserted at the tail, and retrieved from the head of the queue.
* Once created, the capacity of the queue cannot be changed.
* It supplies **blocking insertion and retrieval operations**.
* It does not allow NULL objects.
* ArrayBlockingQueue is **thread safe**.
* The Iterator provided in method **iterator()** traverse the elements in order from first (head) to last (tail).
* ArrayBlockingQueue supports an optional **fairness policy** for ordering waiting producer and consumer threads. With fairness set to true, the queue grants threads access in FIFO order.

|  |
| --- |
| ArrayBlockingQueueExample.java |
| import java.util.concurrent.ArrayBlockingQueue;  import java.util.concurrent.TimeUnit;    public class ArrayBlockingQueueExample  {      public static void main(String[] args) throws InterruptedException      {          ArrayBlockingQueue<Integer> priorityBlockingQueue = new ArrayBlockingQueue<>(5);            //Producer thread          new Thread(() ->          {              int i = 0;              try              {                  while (true)                  {                      priorityBlockingQueue.put(++i);                      System.out.println("Added : " + i);                        Thread.sleep(TimeUnit.SECONDS.toMillis(1));                  }                } catch (InterruptedException e) {                  e.printStackTrace();              }            }).start();            //Consumer thread          new Thread(() ->          {              try              {                  while (true)                  {                      Integer poll = priorityBlockingQueue.take();                      System.out.println("Polled : " + poll);                        Thread.sleep(TimeUnit.SECONDS.toMillis(2));                  }                } catch (InterruptedException e) {                  e.printStackTrace();              } |

LinkedTransferQueue

**Java TransferQueue** is a concurrent blocking queue implementation in which producers may wait for receipt of messages by consumers. **LinkedTransferQueue** class is an implementation of TransferQueue in Java.

## 1. LinkedTransferQueue Features

Let’s note down few important points on the LinkedTransferQueue in Java.

* LinkedTransferQueue is an **unbounded** queue on linked nodes.
* This queue orders elements FIFO (first-in-first-out) with respect to any given producer.
* Elements are inserted at the tail, and retrieved from the head of the queue.
* It supplies **blocking insertion and retrieval operations**.
* It does not allow NULL objects.
* LinkedTransferQueue is **thread safe**.
* The size() method is NOT a constant-time operation because of the asynchronous nature, so may report inaccurate results if this collection is modified during traversal.
* The bulk operations addAll, removeAll, retainAll, containsAll, equals, and toArray are not guaranteed to be performed atomically. For example, an iterator operating concurrently with an addAll operation might view only some of the added elements

|  |
| --- |
| LinkedTransferQueueExample.java |
| LinkedTransferQueue<Integer> linkedTransferQueue = new LinkedTransferQueue<>();    linkedTransferQueue.put(1);    System.out.println("Added Message = 1");    Integer message = linkedTransferQueue.poll();    System.out.println("Recieved Message = " + message); |

CopyOnWriteArrayList

It is fail safe iteration class.

## CopyOnWriteArrayList Features

The important things to learn about **Java CopyOnWriteArrayList** class are:

* CopyOnWriteArrayList class implement List and RandomAccess interfaces and thus provide all functionalities available in ArrayList class.
* Using CopyOnWriteArrayList is costly for update operations, because each mutation creates a cloned copy of underlying array and add/update element to it.
* It is thread-safe version of ArrayList. Each thread accessing the list sees its own version of snapshot of backing array created while initializing the iterator for this list.
* Because it gets snapshot of underlying array while creating iterator, it **does not throw ConcurrentModificationException**.
* Mutation operations on iterators (remove, set, and add) are not supported. These methods throw UnsupportedOperationException.
* CopyOnWriteArrayList is a concurrent replacement for a **synchronized List** and offers better concurrency when iterations outnumber mutations.
* It allows duplicate elements and heterogeneous Objects (use generics to get compile time errors).
* Because it creates a new copy of array everytime iterator is created, **performance is slower** than ArrayList.

|  |
| --- |
| CopyOnWriteArrayList Example |
| CopyOnWriteArrayList<Integer> list = new CopyOnWriteArrayList<>(new Integer[] {1,2,3});    System.out.println(list);   //[1, 2, 3]    //Get iterator 1  Iterator<Integer> itr1 = list.iterator();    //Add one element and verify list is updated  list.add(4);    System.out.println(list);   //[1, 2, 3, 4]    //Get iterator 2  Iterator<Integer> itr2 = list.iterator();    System.out.println("====Verify Iterator 1 content===="); |

CopyOnWriteArraySet

**Java CopyOnWriteArraySet** is a [thread-safe](file:///D:\websites\howtodoinjava.com\java\multi-threading\what-is-thread-safety\index.htm) variant of [**HashSet**](file:///D:\websites\howtodoinjava.com\java\collections\java-hashset\index.htm) which uses a underlying CopyOnWriteArrayList for all of its operations.

## CopyOnWriteArraySet Features

The important things to learn about Java CopyOnWriteArraySet class are:

* As normal set data structure, it does not allow duplicates.
* CopyOnWriteArraySet class implement Serializable interface and extends AbstractSet class.
* Using CopyOnWriteArraySet is costly for update operations, bacause each mutation creates a cloned copy of underlying array and add/update element to it.
* It is thread-safe version of HashSet. Each thread accessing the set sees its own version of snapshot of backing array created while initializing the iterator for this set.
* Because it gets snapshot of underlying array while creating iterator, it **does not throw ConcurrentModificationException**.
* Mutation operations on iterators are not supported. These methods throw UnsupportedOperationException.
* CopyOnWriteArraySet is a concurrent replacement for a **synchronized Set** and offers better concurrency when iterations outnumber mutations.
* It allows duplicate elements and heterogeneous Objects (use generics to get compile time errors).
* Because it creates a new copy of underlying array everytime iterator is created, **performance is slower** than HashSet.

|  |
| --- |
| CopyOnWriteArraySet Example |
| CopyOnWriteArraySet<Integer> set = new CopyOnWriteArraySet<>(Arrays.asList(1,2,3));    System.out.println(set);    //[1, 2, 3]    //Get iterator 1  Iterator<Integer> itr1 = set.iterator();    //Add one element and verify set is updated  set.add(4);  System.out.println(set);    //[1, 2, 3, 4]    //Get iterator 2  Iterator<Integer> itr2 = set.iterator();    System.out.println("====Verify Iterator 1 content===="); |

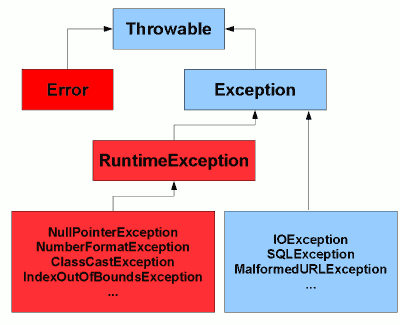
Collection Sorting

Refer in how to do in java

Date

Excxeption

Exception is un expected event that happened during execution of program, that may lead to ubnormal termination of program.



Try, catch, finally

Try is the block it contains statements that may raise the exception during execution of program.

Cache is the block , it can capable of catch and handle the exeption

Finally is the block , it is used to relinquish the resources when exeption is raised.

Checked vs Unchecked

Exception which are checked at Compile time called Checked Exception.  If in our code if some of method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword.

1. IOException
2. SQLException
3. DataAccessException
4. ClassNotFoundException
5. InvocationTargetException
6. MalformedURLException

Unchecked Exception in Java is those Exceptions whose handling is NOT verified during Compile time. These exceptions occurs because of bad programming. The program won’t give a compilation error. All Unchecked exceptions are direct sub classes of RuntimeException class.

* NullPointerException
* ArrayIndexOutOfBound
* IllegalArgumentException
* IllegalStateException

Throw vs throws

Throw keyword is used to throw new exception when program behave abnormally.

Throws keyword is used in method head to declare exception.

Asynchronous and Synchronous

HandleNullPointerException

Exception handling

the process of catch and handle the created exception is called exception handling.

Custom exception

If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

By the help of custom exception, you can have your own exception and message.

class InvalidAgeException extends Exception{

 InvalidAgeException(String s){

  super(s);

 }

}

Generics

Generic is a programming feature. Using generic concept we can provide common implementation to class or interface to handle different types of data types.

interface DemoInterface<T1, T2>

{

   T2 doSomeOperation(T1 t);

   T1 doReverseOperation(T2 t);

}

Learn generic wild cards

Types of Generics

[Types of Generics?](file:///D:\websites\howtodoinjava.com\java\generics\complete-java-generics-tutorial\index.htm#generic_types)

[i) Generic Type Class or Interface](file:///D:\websites\howtodoinjava.com\java\generics\complete-java-generics-tutorial\index.htm#member_types)

[ii) Generic Type Method or Constructor](file:///D:\websites\howtodoinjava.com\java\generics\complete-java-generics-tutorial\index.htm#method_types)

 public Dimension(T length, T width, T height)

   {

      super();

      this.length = length;

      this.width = width;

      this.height = height;

   }

Object life cycle

Serialization

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory

Serializable vs Externalizable

Externalizable is an interface. It is efficiant method than serializable here we can provide our own decode and encode logics through writeExternal() and readExternal().

public void writeExternal(ObjectOutput out) throws IOException {

    //We are not storing the field 'doNotStoreMe'

    out.writeInt(fieldOne);

    out.writeUTF(fieldTwo);

    out.writeBoolean(fieldThree);

}

public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {

    fieldOne = in.readInt();

    fieldTwo = in.readUTF();

    fieldThree = in.readBoolean();

}

Serialization Read object and write object

 private void readObject(ObjectInputStream aInputStream) throws ClassNotFoundException, IOException

    {

        firstName = aInputStream.readUTF();

        lastName = aInputStream.readUTF();

        accountNumber = aInputStream.readInt();

        dateOpened = new Date(aInputStream.readLong());

    }

    private void writeObject(ObjectOutputStream aOutputStream) throws IOException

    {

        aOutputStream.writeUTF(firstName);

        aOutputStream.writeUTF(lastName);

        aOutputStream.writeInt(accountNumber);

        aOutputStream.writeLong(dateOpened.getTime());

    }

String class (read Example in how to do in java)