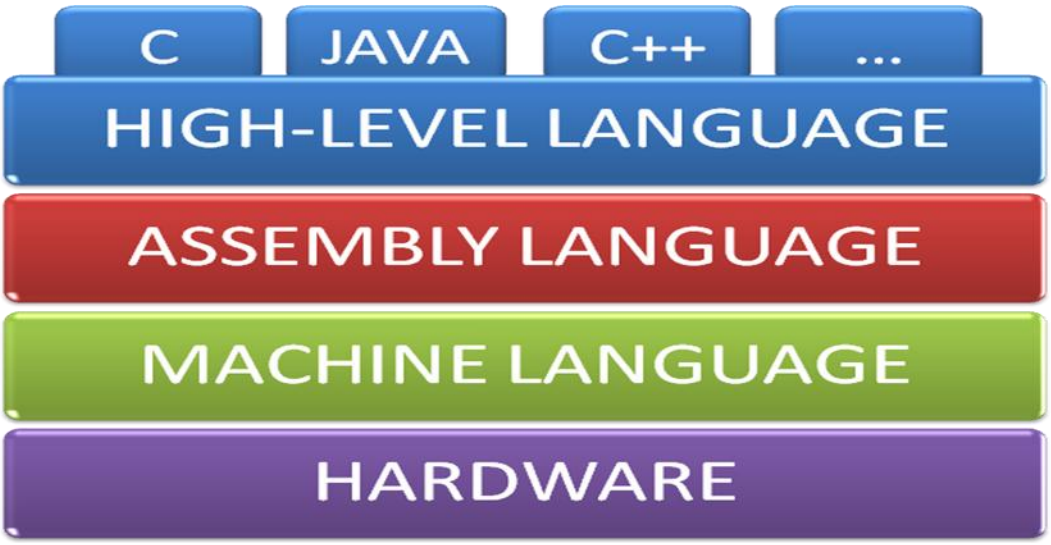
WHAT IS JAVA?

Java is a high level **programming language** and a **platform**.

PROGRAMMING LANGUAGE

A vocabulary and set a grammatical rules for instructing a computer to perform specific task.



Machine Language:

- Machine language consists of only numbers (0 & 1)
- It is understandable by the computer (CPU)
- No matters, in which language we write a program. It has to be converted to the machine language before it is send to CPU

Assembly Language:

- It is similar to machine language
- In addition to machine language, it has a concept of variable – to store numbers
- Program which convert the **Assembly Language** to **Machine Language** is called **Assembler**

High Level Language:

- It has a set of vocabulary (keywords) and grammatical rules to instruct a computer to perform a specific task

- Program which convert the **High Level Language** to **Machine Language** is called a **Compiler**
- A **Compiler** goes through the entire code and then translates it into machine code.
- On the other hand, **Interpreter** reads one statement at a time and translates it into machine code.

PLATFORM

Any hardware or software environment in which a program runs is called a **platform**.
Since Java has its own runtime environment (**JRE**), so it is also called a platform.

TYPE OF JAVA APPLICATIONS

There are mainly 4 types of applications which can be created using java

1. STANDALONE APPLICATION

Standalone applications are desktop applications which need to be installed or deployed in each machine before using it.

Java provides frameworks like AWT, Java Swing, Java FX to develop such applications with ease.

2. WEB APPLICATION

Web Applications are applications running on the web server, and can only be interacted using http protocol.

The web applications are deployed in **Web Server**.

Web server has only web container.

3. ENTERPRISE APPLICATION

Enterprise applications are applications running on the application server, which can be interacted by http protocol as well as RMI etc.

Enterprise application has web container, EJB container, Transaction Services, Connection Pooling etc features apart from just web container.

Enterprise application runs on **Application Server**.

4. MOBILE APPLICATION

Mobile applications are applications or apps running in mobile device.

HISTORY OF JAVA

- **James Gosling, Mike Sheridan** and **Patrick Naughton** initiated the Java language project in June 1991
- Java was originally designed for interactive television, but it was too advanced for the digital television at that time
- Initially it was called OAK after an oak tree stood outside Gosling's office
- Later the project was called Green
- Finally, it was renamed as **Java,** from Java Coffee
- Sun Microsystems released the first public implementation as Java 1.0 in 1995
- It promised "Write Once, Run Anywhere" (WORA)
- Oracle acquires Sun Microsystem in 2009-2010
- Currently, Java 8 is the latest version released in March, 2014

PRINCIPLES

The Java language is developed based on the following principles:

1. Simple, Object-Oriented and Familiar
2. Robust and Secure
3. Architecture-neutral and portable
4. High Performance
5. Interpreted, threaded and dynamic

CONFIGURE JAVA

- DOWNLOAD JDK
  - Download JDK from the given url (http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html)
- ENVIRONMENT VARIABLE SETTING
  - JAVA_HOME
  - Path
- RUN A DEMO HELLO WORLD PROGRAM
  - javac
  - java

WHAT ARE JDK, JRE AND JVM?

JDK:

Java Development Kit (JDK) contains tools to develop, compile, debug and execute a java program. It includes JRE (Java Runtime Environment).

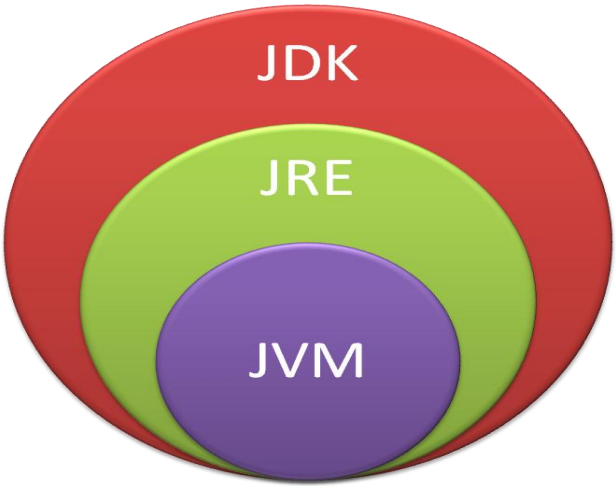
It contains all the necessary tools, files and library to **develop and debug** a java program which is not a part of JRE.

JRE:

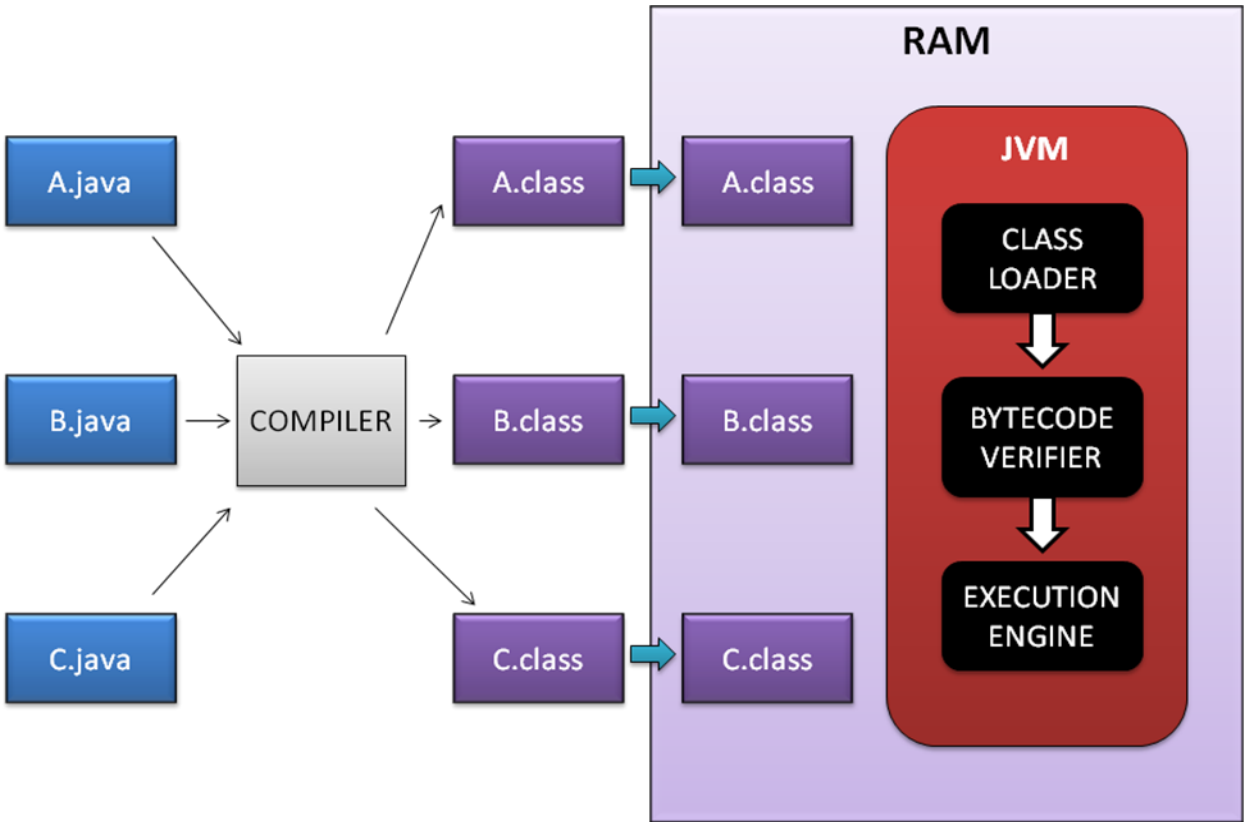
Java Runtime Environment (JRE) contains JVM, class libraries and other supporting files.
To run any java class files, JRE must be installed in the system

JVM:

Java Virtual Machine (JVM) is the actual place where java classes are loaded, verified, transform class file into machine format and execute the program. The class libraries and support files necessary for JVM to perform it tasks are supplied by JRE.



How Java program compiles and run



- The java file is compiled by the compiler and will be converted into bytecode (.class file).
- The bytecode is loaded into RAM by the CLASS LOADER present in JVM
- Once the bytecodes are loaded, the BYTECODE VERIFIER will verify the bytecode for security reasons
- Once the bytecode is verified, it will be converted to machine code and will be processed by the Processor.
- This process is also call Just In Time Compiler (JIT) and due to this reason the Java performance is little slower than other language like C, C++ etc.

Object, Class And Package

This part describes the base concepts we should know before we deep drive into Java Programming Language.

What is an Object?

Object is the key to understand object-oriented technology.

Object has two characteristics:

1. State (Fields or Variables)
2. Behavior (Functions or Methods)

We can say *an **Object** is a software bundle of related state and behavior*.

We can compare the software objects with real world objects.
For example, dog, desktop lamp, car, bicycle etc.

Dog has state (name, breed, color) and behavior (barking, waggling tail).
Desktop has state (on and off) and behavior (turning on and turning off).

WHAT IS A CLASS?

In the real world, we will often find many individual objects of same kind. There are thousands of Hyundai cars of same make and model.

So, we can say a ***Class** is a blueprint from which individual objects are created*

For Eg,

```
class Car {
    int speed = 0;
    int gear = 0;

   void applyBrake (int decrement) {
      speed = speed – decrement;
    }

   void changeGear (int newValue) {
      gear = newValue;
    }

}
```

How we can create objects from the Car class:

```
Car myCar1 = new Car();  // myCar1 object  will be created.
Car myCar2 = new Car(); // new myCar2 object will be created.
```

WHAT IS AN INTERFACE?

An interface is a way to interact an object with the external world. Objects define and expose methods to interact with the outside world.

For eg, The power button infront of the television is an interface to interact between you and the electric wiring on the other side of its plastic casing.

```
interface Car {
```

```
    void changeGear();

    void increaseSpeed();

}
```

WHAT IS A PACKAGE?

A package is a namespace that organizes a set of classes and interfaces.

Conceptually, we can think of packages as being similar to different folders on our computer.

DECLARATIONS AND ACCESS CONTROL

In Java, there are four (4) access controls but three (3) access modifiers (public, protected and private).

Modifiers are divided into 2 categories. They are

- ACCESS Modifiers (public, protected and private)
- NON ACCESS Modifiers (including strictfp, final and abtract)

CLASS DECLARATIONS AND MODIFIERS

When we say class A has access to class B that means the class A can do the following three things:

- Can create an instance of Class B
- Can extends class B
- Can access methods and variables of class B, depending on the access modifiers of those methods and variables

CLASS LEVEL ACCESS MODIFIER

At class level, only 2 access modifiers can be used:

- default (package-level, when no access modifier is used)
- public

Default Access

A class with default access has no modifiers preceding its declaration.

- The class having default access will be accessed by any class present in the **same package**.
- We can think of default access as **package-level access**.

Eg.

```
    class Car {
    }
```

The class car will be accessed by all the other classes present inside the same package.

Public Access

A class with public access has a public modifier preceding its declaration.

- The class having public access can be accessed by any class present in any package.
- All classes in the **Java Universe (JU)** have access to public class

Eg,

```
public class Car {
}
```

CLASS LEVEL NON ACCESS MODIFIER

At the class level, 3 non access modifiers are used

- final
- abstract
- strictfp

Final Classes (Non Access Modifier)

When we use final keyword in the class declaration, it means that the class cannot be extended. We cannot create a subclass of that class.

It breaks the object-oriented notion of inheritance. So, we should create a final class only when we are sure that none of its method will be ever overridden.

In Java, there are many classes which are final.
For eg. One such class is String which is a final class and it can't be extended or subclass.

```
public final class Car {
        public void importantMethod () {
        }
}
```

Abstract Classes (Non Access Modifier)

When we us abstract keyword in the class declaration, it means that class can't be instantiated. Its sole purpose it to get extended or subclass. It is opposite to final modifier.

If a method in a class is declare abstract then entire class needs to be declared as abstract.

For eg,

```
public abstract class Car {
}
```

We can't instantiate the car class now as it is declared abstract. So, it will throw compilation error when we try to create a object of Car.

```
Car myCar3 = new Car();  // Not possible, we will get compilation error
```

Strictfp Classes (Non access modifier)

Strictfp means strict floating point.

Generally, the floating point representation varies across different OS. So, to define a unique or same floating point representation across different OS's strictfp is used.

Marking a class as strictfp means that any method code in the class will conform to the IEEE 754 standard rules for floating points. Without that modifier, floating points used in the methods might behave in a platform-dependent way.

INTERFACE DECLARATIONS AND MODIFIERS

When we define an interface, we are actually defining a contract for what a class can do, without saying anything how the class will do.

The class which implements the interface has to write the code for all the methods defined in the interface (contract).

An **abstract** class can defined **both abstract and non abstract methods**, but an **interface** can have **only abstract method.** There are strict rules to define the variables and methods in interface. Those are:

- All interface **methods** are implicitly **public and abstract** even we defined it or not.
- All **variables** in an interface must be **public static and final**. In other words, interfaces declare only **CONSTANTS not instant variables**
- Interface methods must not be **static**.
- Because interface methods are always abstract they can't be final, strictfp or native. We can't use any modifiers with **abstract methods** except public and protected.
- An interface can extends one or more interfaces.
- An interface can only extends interface
- An interface cannot implements another interface or class
- An interface must be defined with the keyword **interface**

The following is the legal interface declaration:

```
public abstract interface Car {}
or
public interface Car {}
```

The abstract keyword is reluctant here, whether we use abstract or not, interface are implicitly abstract.

Interface Methods:
All interface methods is always **public abstract**, whether we declare it or not.
We can't use protected or private access modifiers in the interface methods

```
public interface Car {
        void move();
}
```

If we define the above code, the compiler we see the code at runtime like,

```
public abstract interface Car {
        public abstract void move();
}
```

INTERFACE LEVEL ACCESS MODIFIER

There are 2 access modifiers which are applicable at the interface level. They are

- default Access
- public Access

Default Access

Similar to class (default access)
Please refer Class Level Access Modifier for more details.

Public Access

Similar to class (public access)
Please refer Class Level Access Modifier for more details.

INTERFACE LEVEL NON ACCESS MODIFIER

There is only one non access modifier which is applicable to the interface.

- Abstract

Abstract Interface

Whether we declare it or not in the interface declaration, abstract keyword will be implicitly added by the compiler at runtime.

As the interface methods are implicitly abstract, so as per the abstract rule the class or interface should be declared abstract.

CLASS MEMBERS AND MODIFIERS

Class Members refers to methods or instance variables in a class.

Let's discuss about the access and non-access modifiers applicable to class members.

CLASS MEMBER ACCESS MODIFIERS

Unlike class, class members use all the four access controls. They are

- private
- default
- protected
- public

Private Access

When a class member is declared as private, it means that class member is **only accessible or visible from within the class, where it is defined**. Other class won't be able to access any class members which are declared private.

For eg,

```
public class Car {
        private int privateVariable;

        private void privateMethod () {
                System.out.println ("We are inside Private Method");
        }

        private void usePrivateMethod() {
                privateMethod();
        }
}
```

Default Access

When a class member is declared as default, it means that class member is **accessible or visible from another class within the same package**.
It won't be visible or accessible from the class which is not present in the same package.

For eg,

```
public class Car {
        int defaultVariable;

        void defaultMethod () {
                System.out.println ("We are inside Default Method") ;
        }
}
```

The class member defaultVariable and defaultMethod will be accessible from any class within the same package.

Protected Access

When a class member is declared protected, it means that class member is **accessible or visible to other class present in the same package as well as it is accessible via inheritance**.

For eg,

```
package com.mycompany.package1;
public class Car {

        protected int protectedVariable;

        protected void protectedMethod () {
                System.out.println ("We are inside Protected Method");
        }
}
```

To access the protected members within the same package we can use the same way as default members. But to access the protected member from another package, we need to extend the class present in another package and then we have to use it.

For eg,

```
package com.mycompany.package2;
 public class BMW extends Car {
        private void testProtectedMethodAccess () {
                Sysout.out.println("We are calling the protected method via inheritance ");
                protectedMethod();
        }
}
```

Public Access

When a class member is declared as public, it means that class member is **accessible or visible to all the other class present anywhere in the Java universe**.

For eg,

```
 public class Car {
        public int publicVariable;

        public void publicMethod () {
                System.out.println("We are inside public method");
        }
}
```

Visibility	Public	Protected	Default	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No

From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes, only via inheritance	No	No
From any non-subclass class outside the same package	Yes	No	No	No

CLASS MEMBER NON ACCESS MODIFIERS

Let us first discuss about the non access modifiers applicable to methods.

- final
- abstract
- static
- synchronized
- native
- strictfp

Final Method
The method declared as final can't be override in the subclass.

Abstract Method
The method declared as abstract doesn't have the implementation. It needs to be implemented in the concrete class which extends the Abstract class.

Static Method
The static keyword is used to create a method that will exist independently of any instances created for the class.

Synchronized Method (Only Method and block)
When a method is declared as synchronized it means that method is accessed by only one thread at a time.

Native Method (Only Method)
The native method indicates that the method is implemented in platform dependent code, generally C.

Strictfp Method
When a method is declared as strictfp, it means that any floating number will be conformed to IEEE 754 standard.

Let us discuss the non access modifier applicable to variables

- final
- static
- volatile (Only Variable)
- transient (Only Variable)

final Variable

When a variable is declared as final, it means that value in that variable can't be changed or modify. This holds good for primitive type.

When a reference variable is declared as final, it means that it can't be reassigned to refer to a different object. The data within the object can be modified, but the reference variable cannot be changed.
There are no final objects, only final reference.

Static Variable

The static keyword is used to create a variable that will exist independently of any instances created for the class

Volatile Variable (Only Variable)

When a variable is declared as volatile, it means the value assigned to it will be modified by different threads. The value of it will never be cache thread-locally, all the changes will be stored in main memory.

Transient Variable (Only Variable)

When a variable is declared as transient, it means that variable will be ignored by the JVM when we attempt to serialize the object containing it.

Local Variables	Variables (non local)	Methods	Class
• final	• private • <default> • protected • public • final • static • transient • volatile	• private • <default> • protected • public • final • static • abstract • strictfp • synchronized • native	• <default> • public • final • abstract • strictfp

OBJECT ORIENTATION

ENCAPSULATION

The ability to change the implementation of our code without breaking the code of others is the key benefit of **encapsulation.**

Encapsulation also means hiding the implementation details and provides a mean to use those implementations. Generally, we use interface to expose methods which can be called by other implementation class.

If we want maintainability, flexibility and extensibility, we must incorporate encapsulation in our code:

- Keep instance variables protected using private modifier
- Defined getter and setter methods to access those instance variables

For eg,

```
class Test {
        private int positiveNumber;

        public int getPositiveNumber() {
                return positiveNumber;
         }

         public void setPositiveNumber(int positiveNumber) {
                this.positiveNumber = positiveNumber; //We can write validation logic here
        }

}
```

INHERITANCE

Inheritance is a concept in which, the child acquires the behavior and state of its parent.

The most important reasons to use inheritance are as follows:

- To promote code reuse
- To use polymorphism

It is always a good idea to define the common methods in the parent class.
The child will automatically inherit those methods, instead of duplicating the code in each class.

If we are not satisfied with the implementation of parent class, then we can modify those methods in the child class.

IS-A AND HAS-A

IS-A

It is a way of saying, "**this thing is a type of that thing**"
IS-A is based on class inheritance or interface implementation.
In simple words,

- If class A extends class B, then class A is-a type of class B
- If class A implements interface C, then class is-a type of class C

HAS-A

When a class has a reference of another class within it, without the use of inheritance (**extends**) or interface implementation (**implements**) then that class Has-A relations with the reference class.

POLYMORPHISM

Any Java object that can pass more than one IS-A test can be considered polymorphic.
Other than objects of type O, all Java objects are polymorphic in nature as it pass the IS-A test for their own type and for class Object.

Polymorphic method invocations apply only to INSTANCE METHODS. We can always refer to an object with more general reference variable type (a superclass or interface), but at runtime, the only things that are dynamically selected based on actual object (rather than the reference type) are instance methods, Not static methods, Not variable not Overloaded methods.

Only overridden instance methods are dynamically invoked based on the real object's type
(at runtime by the JVM)

POLYMORPHISM IN OVERLOADED AND OVERRIDDEN METHODS

```
public class Animal {
        public void eat() {
                System.out.println (" Eat Method in Animal");
        }
}

public class Horse extends Animal {
        public void eat() {
                System.out.println (" Eat Method in Horse");
        }

        public void eat (String s) {
                System.out.println ("Horse eating "+s);
```

```
        }
}
```

Method Invocation Code	Result
Animal a = new Animal(); a.eat();	Eat Method in Animal
Horse h = new Horse(); h.eat();	Eat Method in Horse
Animal ah = **new Horse ();** ah.eat();	Eat Method in Horse **Polymorphism** works – the actual object type (Horse) method is called
Horse he = new Horse(); he.eat ("Apples");	Horse eating Apples The **overloaded** method is invoked
Animal a2 = new Animal (); a2.eat("treats");	Compiler error!
Animal ah2 = new Horse(); ah2.eat ("Carrots");	Compiler error!

OVERLOADED V/S OVERRIDDEN

	Overloaded Method	Overridden Method
Arguments	Must Change	Must not change (including order)
Return Type	Can Change	Can't change expect for covariant returns
Exceptions	Can Change	Can reduce or eliminate. Must not throw new or broader checked exceptions.
Access	Can Change	Must not make more restrictive (can be less restrictive)

*Invocation	Reference type determines which overloaded version is selected	Actual Object type determines which method is selected
Happens at	Compile time	Runtime

STACK AND HEAP

In Java the memory will be divided into two parts:

- STACK
- HEAP

STACK

The stack is the place where any ***primitive values, references to objects, and methods*** are stored.

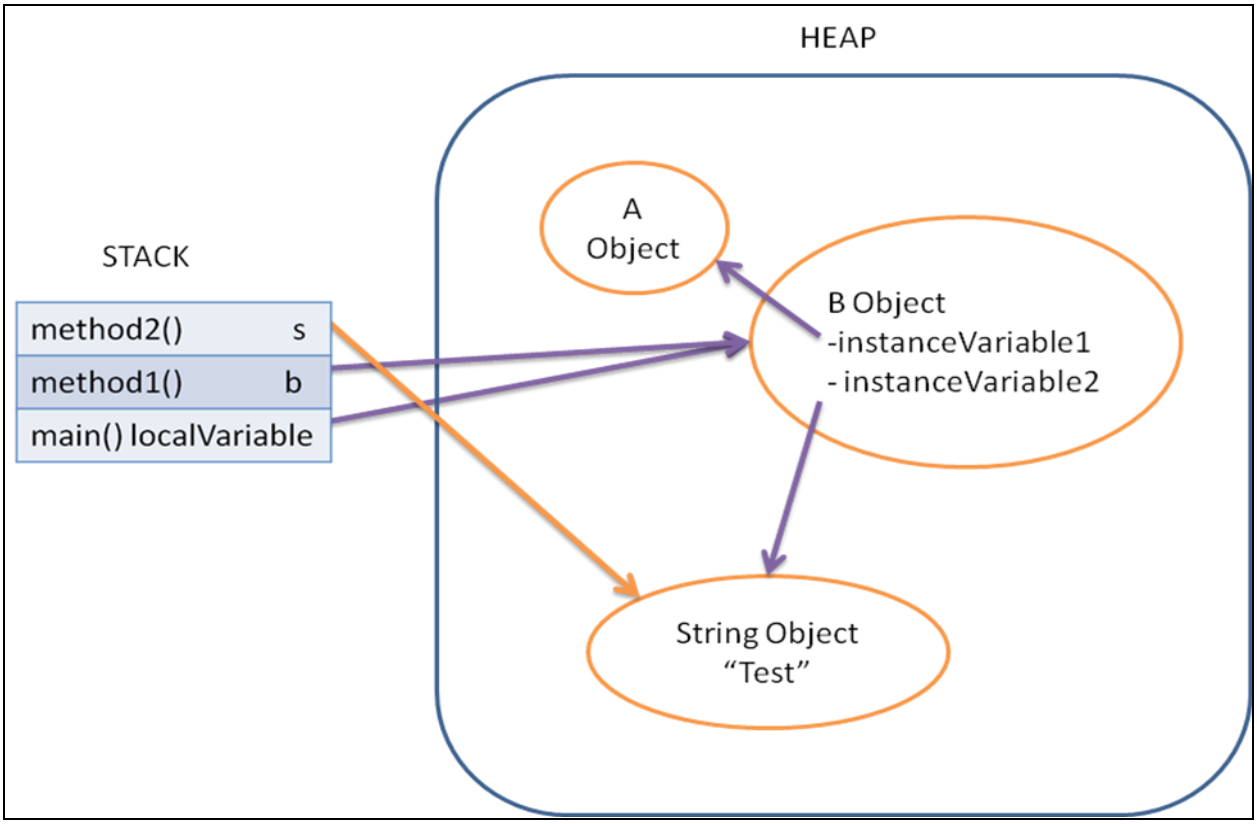
The lifetime of variables on the stack is governed ***by the scope of the code***. The scope is usually defined by an area of code in curly brackets, such as a method call, or for or while loop. Once the execution has left that scope, those variables declared in the scope are removed from the stack.

When the stack memory fills up, and eventually any more method calls will not be able to allocate their necessary variables. This results in a ***StackOverflowError***.

For example:

```
class A {}

public class B {
   A instanceVariable1; //Instance Variable: instanceVariable1
   String instanceVariable2; //Instance Variable: instanceVariable2

   public static void main(String[] args) {
      B localVariable; //Local Variable: localVariable
      localVariable = new B();
      localVariable.method1(localVariable);
   }

   public void method1 (B b) { //Local Variable: b
      b.method2("Test");
   }

   public void method2 (String s) { //Local Variable: s
      instanceVariable2 = s;
   }
}
```

Instance Variables	*Local Variables*	*Methods*	*Objects*
instanceVariable1	localVariable	main	new B()
instanceVariable2	b	method1	new A()
	s	method2	String



HEAP

The heap is the place where ***objects and instance variables*** are stored.

The new keyword allocates memory on the Java heap. The heap is the main pool of memory, accessible to the whole of the application. If there is not enough memory available to allocate for that object, the JVM attempts to reclaim some memory from the heap with a garbage collection. If it still cannot obtain enough memory, an **OutOfMemoryError** is thrown, and the JVM exits

The heap is split into several different sections, called generations. As objects survive more garbage collections, they are promoted into different generations. The older generations are not garbage collected as often. Because these objects have already proven to be longer lived, they are less likely to be garbage collected.

Eden Space	***Survivor Space***	***Tenured Generation***	***Permanent Generation (PermGen)***
When the objects are constructed, they are allocated in the Eden Space	If the objects survive a garbage collection, they are promoted to Survivor Space.	If the object live long enough in the Survivor space, then it is allocated to Tenured Generation	The objects reside here are not eligible to be garbage collected, and usually contain an immutable state necessary for the JVM to run, such as Class Definition and String constant pool ***In JDK 8, the PermGen is removed with a new space called Metaspace, which is held in native memory***

GARBAGE COLLECTION

WHAT IS GARBAGE COLLECTION?

Java provides a mean to create a new object, but it doesn't provide the means to destroy an object **explicitly**.

An object is said to be garbage when no active thread can accessed it, i.e unreferenced object.

The process of finding all the unreferenced objects and the mechanism of reclaiming previously allocated memory, so that it can be used for future memory allocations is called **Garbage Collection**.

In Java, whenever a new object is created, usually by a **new** keyword, the JVM allocates an appropriate amount of memory for that objects and the data it holds.

In C and C++, there is no auto-mated process to reclaim the memory. Manually, we have to call the malloc and free function to release the allocated memory to unreferenced objects.

ALGORITHMS USED BY GARBAGE COLLECTOR

We might hear that garbage collector uses mark and sweep algorithm, and for any given Java implementation that might be true, but the **Java specification doesn't guarantee any particular implementation**.

MARK-AND-SWEEP GARBAGE COLLECTION

The mark and sweep algorithm was the first garbage collection algorithm to be developed that is able to reclaim cyclic data structure.

The mark and sweep algorithm is also called tracing algorithm as it traces out the collection of objects **directly or indirectly accessible** by a program.

When we can say an object is directly accessible by a program?
An object is said to be directly accessible by a program, when a local variable or a static variable has a referenced to it. Those variables in term of garbage collection are said to be **roots**.

When we can say an object is indirectly accessible by a program?
An object is said to be indirectly accessible by a program, when a field of some other accessible object has a reference to it.

Note:
When using mark-and-sweep, unreferenced objects are not reclaimed immediately. **Instead, garbage is allowed to accumulate until all available memory has been exhausted**. When that happens, the execution of the program is suspended temporarily while the mark-and-sweep algorithm collects all the garbage. Once all unreferenced objects have been reclaimed, the normal execution of the program can resume.

The mark and sweep algorithm works in two phases.

- Mark
  - In first phase, it finds and marks all the accessible (directly or indirectly) objects
- Sweep
  - It scans through the heap to reclaims the memory allocation of all unmarked objects.
  - At the same time, it changed all the marked objects to unmarked in the preparation for the next invocation of the mark and sweep algorithm.

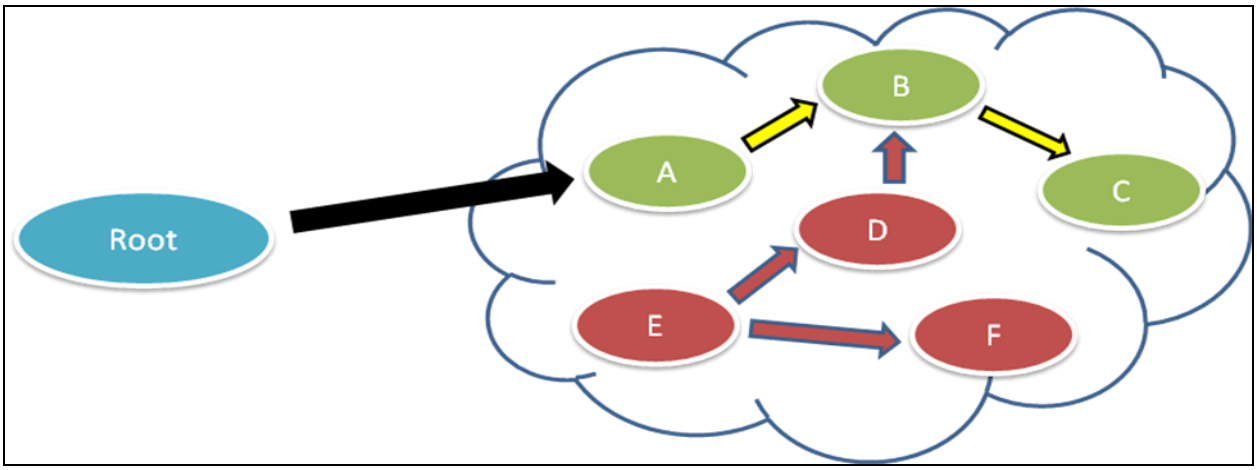


Fig: Java Objects present in Memory (Stack and Heap)

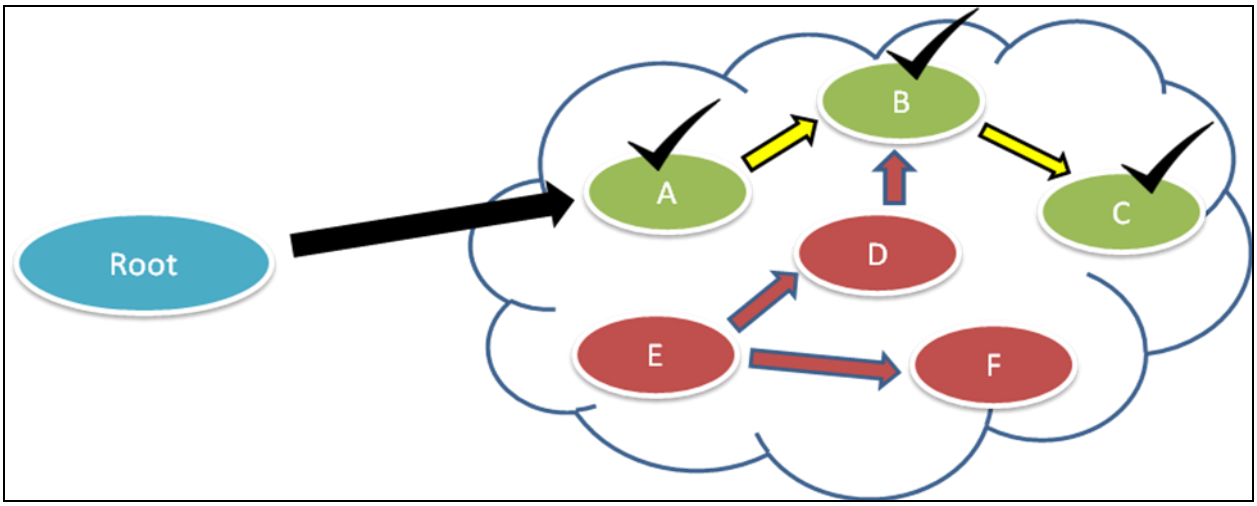


Fig: 1st phase Mark – mark the direct and indirect accessible objects.
Yellow arrow indicates indirect accessible reference objects
Black arrow indicated accessible direct reference objects.

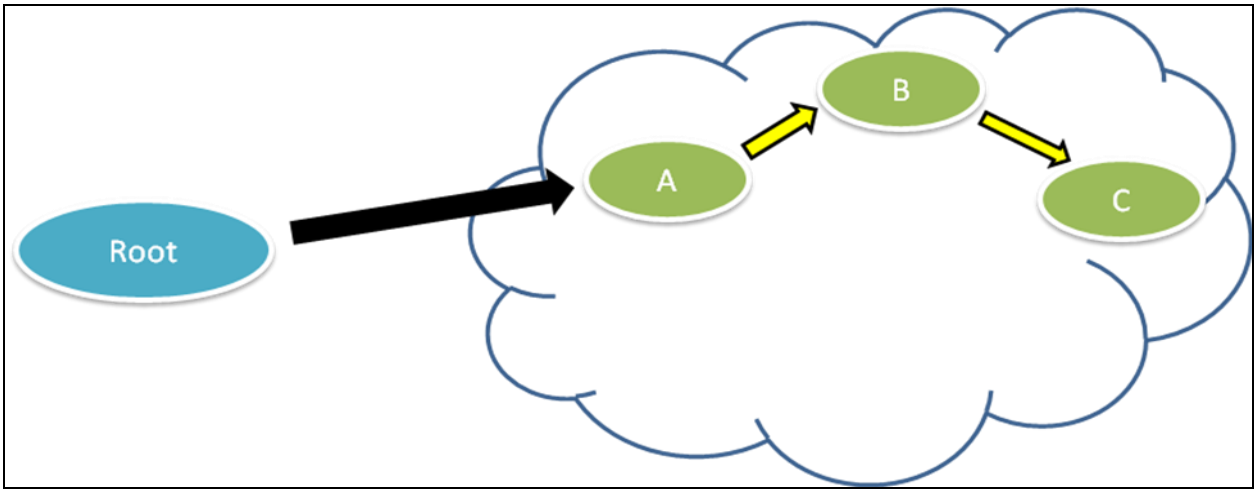


Fig: 2nd phase Sweep – Reclaims the memory allocated to inaccessible objects as well as unmarked the accessible objects for the next invocation of Mark and Sweep algorithm

Advantage:

- Because the mark-and-sweep garbage collection algorithm traces out the set of objects accessible from the roots, it is able to correctly identify and collect garbage even in the presence of reference cycles

Disadvantage:

- The mark-and-sweep approach is the fact that that normal program execution is suspended while the garbage collection algorithm runs. In particular, this can be a problem in a program that interacts with a human user or that must satisfy real-time execution constraints

- The mark-and-sweep algorithm does not address fragmentation. Even after reclaiming the storage from all garbage objects, the heap may still be too fragmented to allocate the required amount of space

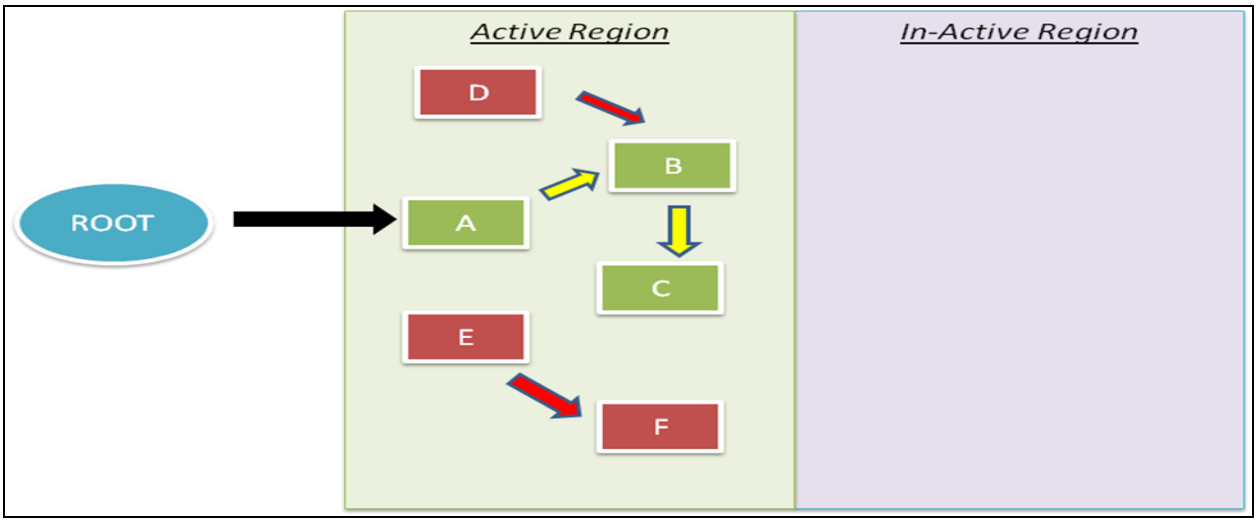
STOP AND COPY GARBAGE COLLECTION

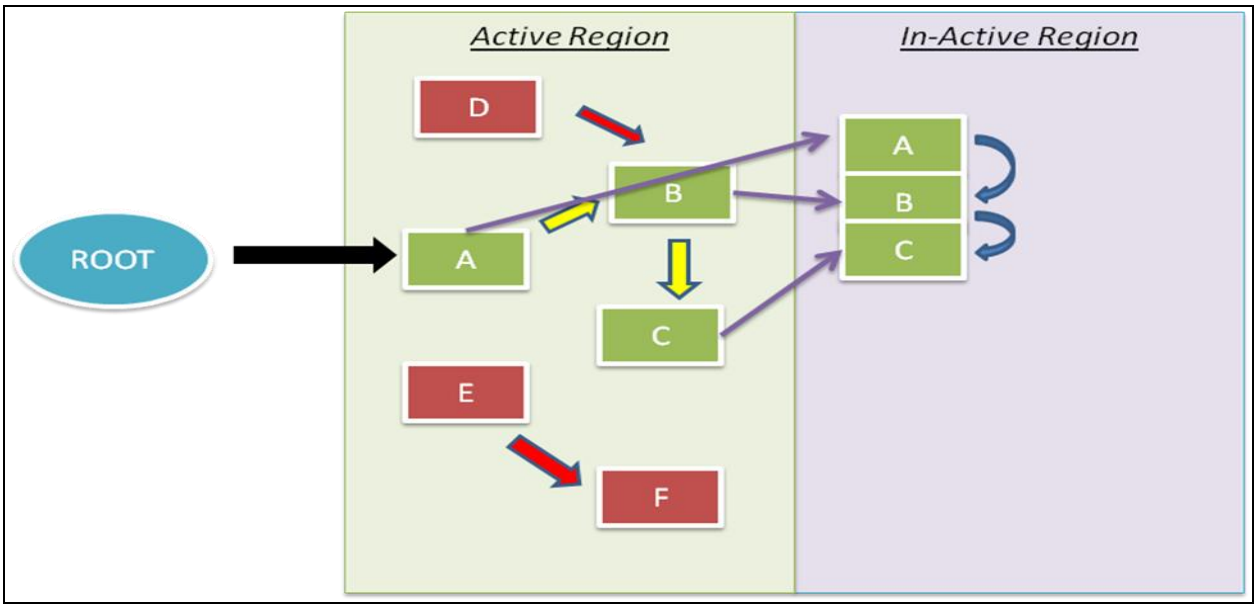
The fragmentation problem of mark and sweep algorithm is taken care in stop and copy algorithm. In Stop and Copy algorithm the heap is divided into 2 regions:

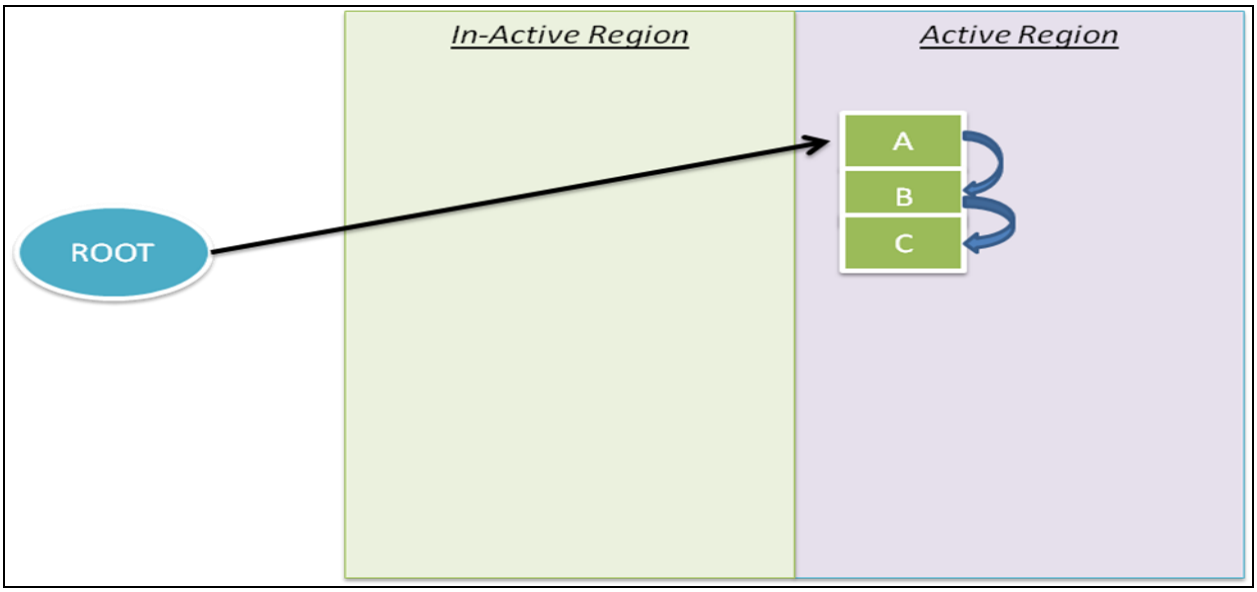
- Active Region
- In-Active Region

When the memory in Active region is exhausted, the program is suspended and the stop and copy algorithm is invoked.
The stop and copy algorithm copy the active/live objects into In-active region and the references are updated to reflect the new locations of the reference objects.

After the copy is completed and the reference values are updated, the active and in-active regions exchange their roles. The storage occupied by the garbage is reclaimed all at once when the active region is in-activate.


Active Region
In-Active Region
ROOT
D
B
A
C
E
F


Active Region
In-Active Region
ROOT
D
B
A
C
E
F
A
B
C



Advantage:
The stop-and-copy algorithm automatically defragments the heap

Disadvantage:
First, the algorithm requires that *all* live objects be copied every time garbage collection is invoked. If an application program has a large memory footprint, the time required to copy all objects can be quite significant

The stop-and-copy is the fact that it requires twice as much memory as the program actually uses. When garbage collection is finished, at least half of the memory space is unused

MARK AND COMPACT GARBAGE COLLECTION

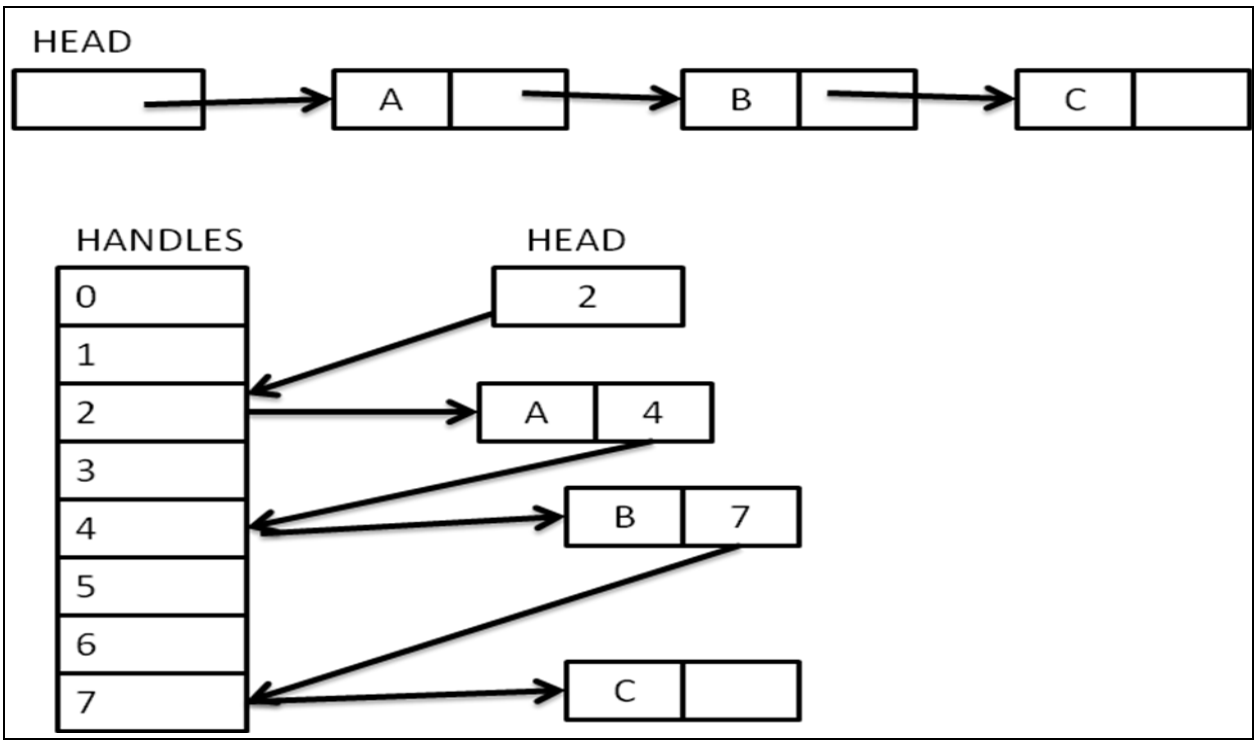
The stop-and-copy algorithm avoids fragmentation at the expense of doubling the size of the heap. This section describes the mark-and-compact approach to garbage collection which eliminates fragmentation without the space penalty of stop-and-copy.

The mark-and-compact algorithm uses the handles in two ways:

- First, the marked flags which are set during the mark operation are stored in the handles rather than in the objects themselves.
- Second, compaction is greatly simplified because when an object is moved only its handle needs to be updated--all other objects are unaffected.

Note:

In this algorithm, an array called handles is used to hold the reference objects. And the object is connected to the handles array using linked list.


HEAD
A
B
C
HANDLES
HEAD
0
1
2
3
4
5
6
7
2
A
4
B
7
C

EXCEPTION HANDLING

The term exception is shorthand for the phrase "exceptional event."

Definition: An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.

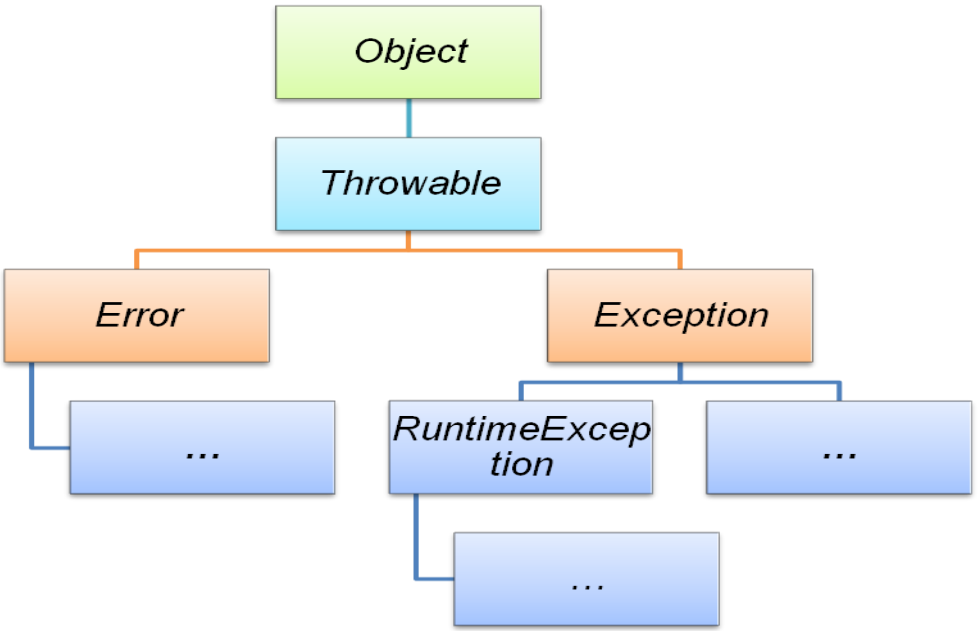
When an error occurs within a method, the method **creates an object** and hands it off to the **runtime system**. The object, called an **exception object**, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called **throwing an exception**.

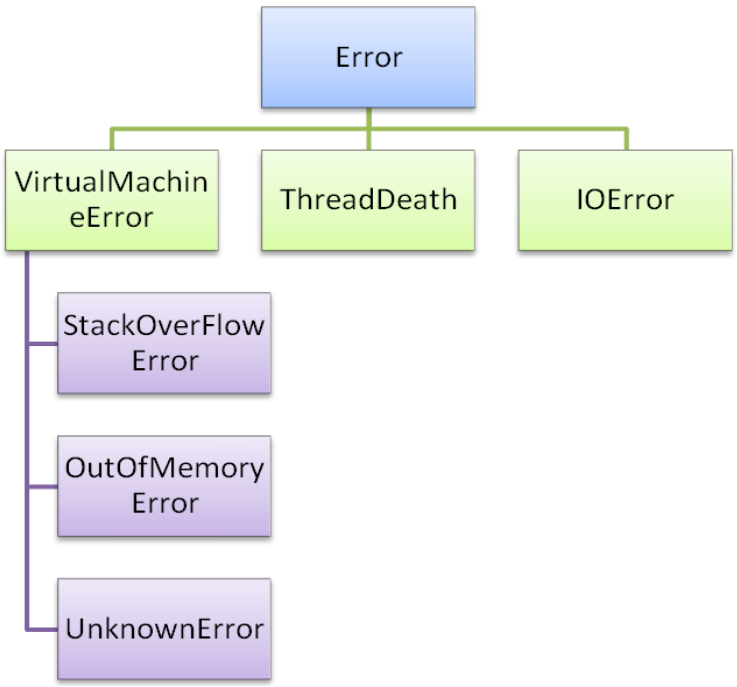
When an exception is thrown, an object of a particular Exception subtype is instantiated and handed to the exception handler as an argument to the catch clause. An actual catch clause looks like this:

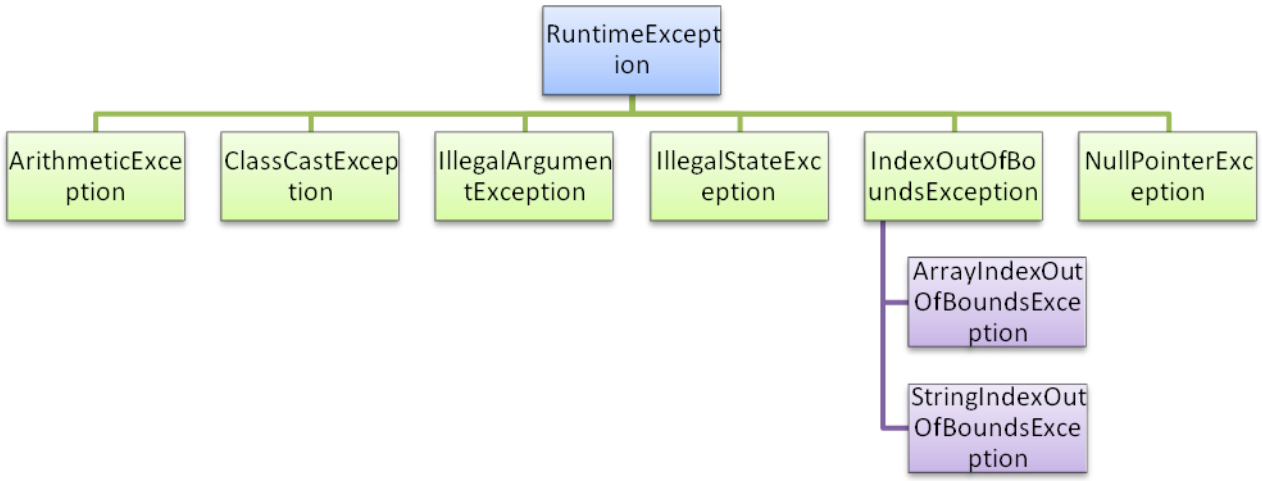
```
try {
        // some code here
} catch (ArrayIndexOutOfBoundsException e) {
        e.printStackTrace();
}
```

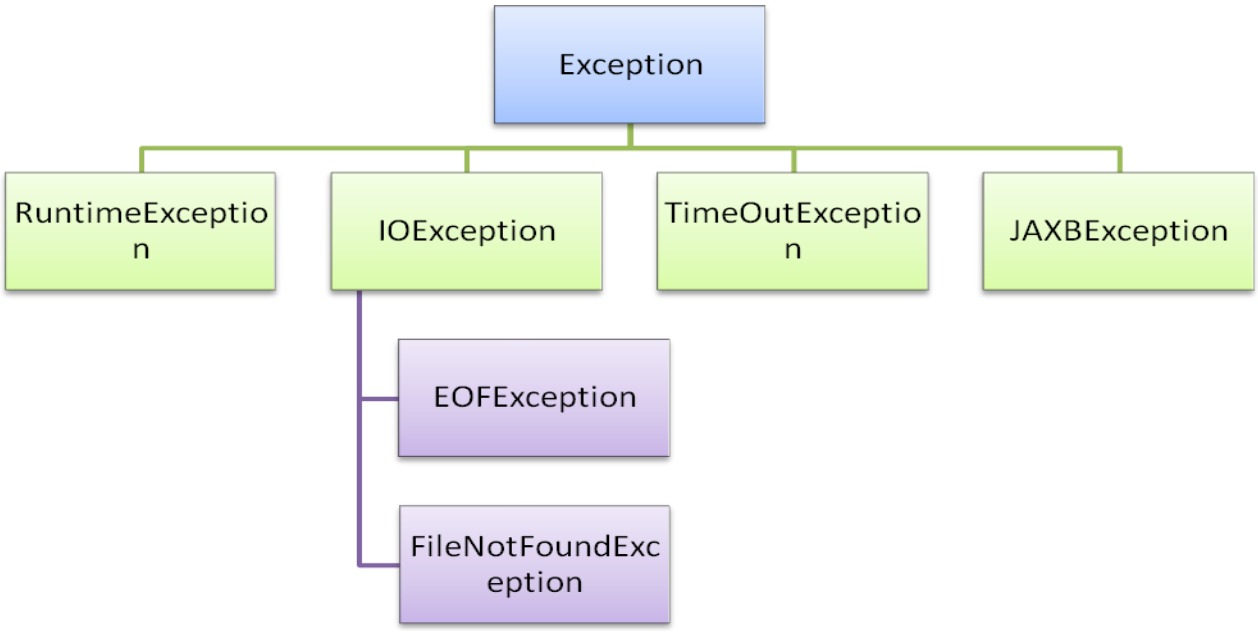
In this example, e is an instance of the ArrayIndexOutOfBoundsException class. As with any other object, you can call its methods.

EXCEPTION HIERARCHY




Error
VirtualMachineError
ThreadDeath
IOError
StackOverFlowError
OutOfMemoryError
UnknownError


RuntimeException
ArithmeticException
ClassCastException
IllegalArgumentException
IllegalStateException
IndexOutOfBoundsException
NullPointerException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException



COMBINATION OF TRY, CATCH AND FINALLY BLOCK

1. ***TRY, CATCH AND FINALLY***

   If try, catch and finally blocks are present then it must follow the below order.
   There may be more than one catch blocks, but the finally should start after the completion of last catch block only.

```
try {

        // Code that might throw exceptions

}catch (Exception ex) {

        // handle thrown exceptions

} finally {

        //Resource cleaned up activities

}
```

2. ***TRY AND CATCH***

   A try and catch blocks can exist without a finally block.

```
try {
```

```
        // Code that might throw exceptions
    }catch (Exception ex) {
        // handle thrown exceptions
    }
```

3. ***TRY AND FINALLY***

A try and finally blocks can exist without a catch block. In that case, we need to throw the checked exceptions using throws keyword.

```
try {
        // Code that might throw exceptions
} finally {
        //Resource cleaned up activities
}
```

TRY-WITH-RESOURCES STATEMENT

The try-with-resources statement is a try statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements **java.lang.AutoCloseable**, which includes all objects which implement **java.io.Closeable**, can be used as a resource.

```
try (BufferedReader br =
        new BufferedReader(new FileReader(path))) {
    return br.readLine();
}
```

DIFFERENCE BETWEEN CHECKED AND UNCHECKED EXCEPTION

DIFFERENCE BETWEEN THROW/THROWS

throw	throws
throw is used to throw an exception explicitly	throws is used to declare an exception
throw is followed by an instance variable	throws is followed by an exception class name
throw is used within the method	throws is used with the method signature
can throw only exception at a time	can declared multiple exceptions
Check exception cannot be propagated using throw only	Check exception can be propagated with throws only

Note:
Sometimes there are requirements where we need to debug the code within jar files and try to understand the flow. In that case, throw play a major role as it will throw the call stack from top to bottom.

COLLECTIONS FRAMEWORK

A collection - is simply an object that groups multiple elements into a single unit.

The collections framework provides a well designed set of interfaces and classes for storing and manipulating groups of data as a single unit, a collection.

Relationships of 4 basic interfaces of Collection framework:

- **Collection** – The Collection interface is a group of objects, with duplicates
- **Set** – The Set interface extends Collection but forbids duplicate
- **List** – The List interface extends Collection, allows duplicates, and introduces positional indexing
- **Map** – The map interface extends neither Set nor Collection. Work with key-value pairs.

COLLECTION INTERFACE

The Collection interface is used to represent any group of objects, or elements.

The interface supports below operations:

- **Altering Operations** – add and remove an element from the collection
  - boolean add(Object element)
  - boolean remove(Object element)
- **Query Operations**
  - int size()
  - boolean isEmpty()
  - boolean contains(Object element)
  - Iterator iterator()
- **Group Operations**
  - boolean containsAll(Collection collection)
  - boolean addAll(Collection collection)
  - void clear()
  - void removeAll(Collection collection)
  - void retainAll(Collection collection)
- **Collection to Array**
  - Object[] toArray()
  - `<T> T[]toArray(T[] a)`

AbstractCollection class

The abstractCollection class provides the basis for the concrete collections framework classes. The abstractCollection class provides implementations for all the methods, except for the iterator() and size() methods, which are implemented in the appropriate subclass. Optional methods like add() will throw an exception if the subclass doesn't override the behavior.

SET INTERFACE

The Set interface extends the Collection interface and, by definition, forbids duplicates within the collection. All the original methods are present and **no new methods are introduced**. The concrete Set implementation classes **rely on the equals() method** of the object added to check for equality.

The three Set implementations are described below:

1. HashSet

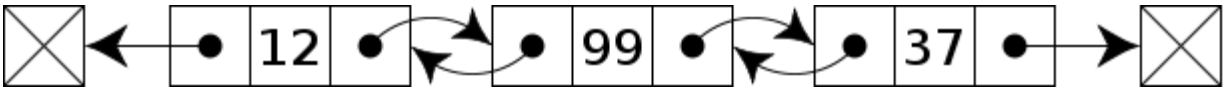
- A HashSet is an unsorted, unordered Set
- Use this class when you want a collection with no duplicates and you don't care about the insertion order.
- This class permits the null element
- This implementation is not synchronized
- The iterators returned by this class's iterator method are fail-fast
- Implements Cloneable, Serializable
- Since 1.2 version

Why fail-fast?
If the set is modified at any time after the iterator is created, in any way except through the iterator's own remove method, the Iterator throws a **ConcurrentModificationException**. Thus, in the face of concurrent modification, the iterator **fails quickly and cleanly**, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future

2. LinkedHashSet

- A LinkedHashSet is an ordered version of HashSet that maintains a **doubly-linked List** across all elements.
- Use this class when you want no duplicates and you care about the insertion order
- This implementation spares its clients from the unspecified, generally chaotic ordering provided by HashSet, without incurring the increased cost associated with TreeSet.
- This implementation is not synchronized
- The iterators returned by this class's iterator method are fail-fast
- Implements Cloneable, Serializable
- Since 1.4 version

A **doubly linked list** whose nodes contain three fields: a value, the link to the next node, and the link to the previous node



3. TreeSet

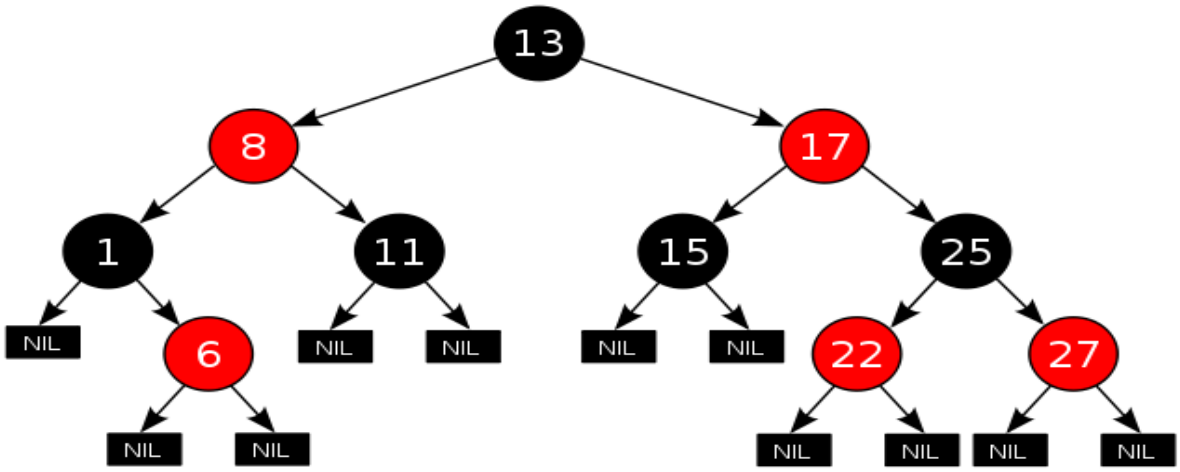
- The TreeSet is one of the two Sorted Collections (the other being TreeMap)
- It used Red-Black Tree Structure
- Guarantees the elements will be in sorted order, ascending order in case of default sorting order
- The Set interface is defined in terms of the equals operation, but a TreeSet instance performs all element comparisons using its compareTo (or compare) method
- Since 1.2 version

Red-Black Tree structure

A red–black tree is a kind of self-balancing binary search tree. Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions.

Balance is preserved by painting each node of the tree with one of two colors (typically called 'red' and 'black') in a way that satisfies certain properties, which collectively constrain how unbalanced the tree can become in the worst case. When the tree is modified, the new tree is subsequently rearranged and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be performed efficiently.

The balancing of the tree is not perfect but it is good enough to allow it to guarantee searching in O(log n) time, where n is the total number of elements in the tree. The insertion and deletion operations, along with the tree rearrangement and recoloring, are also performed in O(log n) time



AbstractSet class

The AbstractSet class overrides the equals() and hashCode() methods to ensure two equal sets return the same hash code.

LIST INTERFACE

The List interface extends the Collection interface to define an ordered collection, permitting duplicates. The interface adds position-oriented operations, as well as the ability to work with just a part of the list.

The positional-oriented operations includes –

- void add(int index, Object element)
- boolean addAll(int index, Collection collection)
- Object get(int index)
- int indexOf(Object element)
- int lastIndexOf(Object element)
- Object remove(int index)
- Object set(int index, Object element)

The List interface also provides for working with a subset of the collection, as well as iterating through the entire list in a position-friendly manner:

- ListIterator listIterator()
- ListIterator listIterator(int startIndex)
- List subList(int fromIndex, int toIndex)

The 3 List implementations are described in the following sections

- ArrayList
- LinkedList
- Vector
    - Stack

1. ArrayList

- An ArrayList is a growable array
- It is an ordered collection (by index), but not sorted
- Use this class when we want fast iterator and fast random access
  (The size, isEmpty, get, set, iterator, and listIterator operations run in constant time)
- Permits null
- This implementation is not synchronized
- The iterators returned by this class's iterator method are fail-fast
- Implements Cloneable, Serializable, RandomAccess
- Since 1.2 version

2. LinkedList

- A LinkedList is ordered by index position, like ArrayList, except that the elements are doubly-linked to one another
- Use this class when you want fast insertion and deletion from the middle of the list and iterate sequentially.
- Permits null
- This implementation is not synchronized
- The iterators returned by this class's iterator method are fail-fast
- Implements Cloneable, Serializable
- As of Java 5, the LinkedList class has been enhanced to implement the java.util.Queue interface. As such, it now supports the common queue methods: peek(), poll(), and offer().
- Since 1.2 version

In addition, LinkedList adds several methods for working with the elements at the ends of the list

- addFirst(E e)
- addLast(E e)
- getFirst()
- getLast()
- removeFirst()
- removeLast()

By using these new methods, you can easily treat the LinkedList as a stack, queue, or other end-oriented data structure.

```
LinkedList stack = ...;
stack.addFirst(element);
Object object = stack.removeLast();
LinkedList queue = ...;
queue.addFirst(element);
```

Object object = queue.removeFirst();

3. Vector

- Vector is a historical collection class that acts like a growable array, but can store heterogeneous data elements
- Vector is basically the same as an ArrayList, but Vector methods are synchronized for thread safety.
- The iterators returned by this class's iterator method are fail-fast
- Implements Cloneable, Serializable, RandomAccess
- Since 1.0 version

3.1 Stack

- The Stack class extends Vector to implement a standard last-in-first-out (LIFO) stack
- Since 1.0 version

AbstractList and AbstractSequencialList Classes

Besides the equals() and hashCode() implementations, AbstractList and AbstractSequentialList provide partial implementations of the remaining List methods. They make the creation of concrete list implementations easier, for random-access and sequential-access data sources, respectively.

MAP INTERFACE

The Map interface is not an extension of the Collection interface. Instead, the interface starts off its own interface hierarchy for maintaining key-value associations. The interface describes a mapping from keys to values, without duplicate keys, by definition.

The interface supports below operations:

- **Altering Operations** – add and remove an key-value from the map
  - Object put(Object key, Object value)
  - Object remove(Object key)
  - void putAll(Map mapping)
  - void clear()

- **Query Operations**
  - Object get(Object key)
  - boolean containsKey(Object key)
  - boolean containsValue(Object value)
  - int size()
  - boolean isEmpty()

- **View Operations**
  - public Set keySet()
  - public Collection values()
  - public Set entrySet()

MAP.ENTRY INTERFACE

The entrySet() method of Map returns a collection of objects that implement the Map.Entry interface. Each object in the collection is a specific key-value pair in the underlying Map.

Let discuss the implementation of Map interfaces:

1. *HASHMAP*
    - Hash table based implementation of the Map interface
    - Permits null key and null values (Equivalent to Hashtable, expects it is unsynchronized and permits null)
    - This implementation is not synchronized
    - The iterators returned by this class's iterator method are fail-fast
    - Implements Cloneable, Serializable
    - Since 1.2 version
2. *LINKEDHASHMAP*
    - Hash table and linked list implementation of the Map interface, with predictable iteration order
    - Permits null elements
    - This implementation is not synchronized
    - The iterators returned by this class's iterator method are fail-fast
    - Implements Cloneable, Serializable
    - Since 1.4 version
3. *TREEMAP*
    - A Red-Black tree based NavigableMap implementation
    - Doesn't Permits null element
    - This implementation is not synchronized
    - The iterators returned by this class's iterator method are fail-fast
    - Implements Cloneable, Serializable
    - Since 1.2 version

THREADS

In Java, a thread means two different things

- **An instance of class java.lang.Thread**
  It like any other Java object having variables and methods, and lives and dies in the heap
- **A thread of execution in a program**
  An individual lightweight process that has its own call stack – one call stack per thread

DEFINING, INSTANTIATING AND STARTING A THREAD

An application that creates an instance of Thread must provide the code that will run in that thread. There are two ways to do that:

- Implementing java.lang.Runnable Interface
- Extending java.lang.Thread

1. *IMPLEMENTING JAVA.LANG.RUNNABLE INTERFACE*

- Implements the Runnable interface
- Runnable interface has only one method run() – write code that needs to be run in a separate thread in a run()
- Every thread of execution begins as an instance of class **Thread**
  Create an instance of class implementing Runnable interface and pass it reference as a constructor args while creating an instance of **Thread**

  Eg,

  ```
  MyThreadRunnable myThreadRunnable = new MyThreadRunnable();
  Thread thread = new Thread(myThreadRunnable);
  ```

- To make the thread a thread of execution, we have to call the start() of class Thread
  Eg, thread.start();

  What happens after we call start() method
  - A new thread of execution starts (with a new call stack)
  - The thread moves from the new state to runnable state
  - When the thread gets a chance to execute, its target run() method will run

Example:

```
public class MyThreadRunnable implements Runnable {

  @Override
  public void run() {
    System.out.println("Run:0: Inside MyThreadRunnable Run Method");
    System.out.println("Run:1: My Thread name is "+Thread.currentThread().getName());
  }

  public static void main(String[] args) {
    System.out.println("Main:0: My Thread name is "+Thread.currentThread().getName());
```

```
        MyThreadRunnable myThreadRunnable = new MyThreadRunnable();
        Thread thread = new Thread(myThreadRunnable);
        System.out.println("Main:1: My Thread name is "+Thread.currentThread().getName());

        thread.run();
        System.out.println("Main:2: My Thread name is "+Thread.currentThread().getName());

        thread.start();
        System.out.println("Main:3: My Thread name is "+Thread.currentThread().getName());
    }

}
```

Output:

```
Main:0: My Thread name is main
Main:1: My Thread name is main
Run:0: Inside MyThreadRunnable Run Method
Run:1: My Thread name is main
Main:2: My Thread name is main
Main:3: My Thread name is main
Run:0: Inside MyThreadRunnable Run Method
Run:1: My Thread name is Thread-0
```

2. *EXTENDING JAVA.LANG.THREAD*

- Extends Thread
- Override the run() method
- To make the thread a thread of execution, we have to call the start() of class Thread

Example:

```
public class MyThread extends Thread {

  @Override
  public void run() {
    System.out.println("Run:0: Inside MyThread Run method ");
    System.out.println("Run:1: MyThread name is "+Thread.currentThread().getName());
  }

  public static void main(String[] args) {
    System.out.println("Main:0: Thread name is "+Thread.currentThread().getName());

    MyThread myThread = new MyThread();
    myThread.setName("Java-Group");
    System.out.println("Main:1: Thread name is "+Thread.currentThread().getName());

    myThread.run();
    System.out.println("Main:2: Thread name is "+Thread.currentThread().getName());

    myThread.start();
    System.out.println("Main:3: Thread name is "+Thread.currentThread().getName());
```

```
  }
}
```

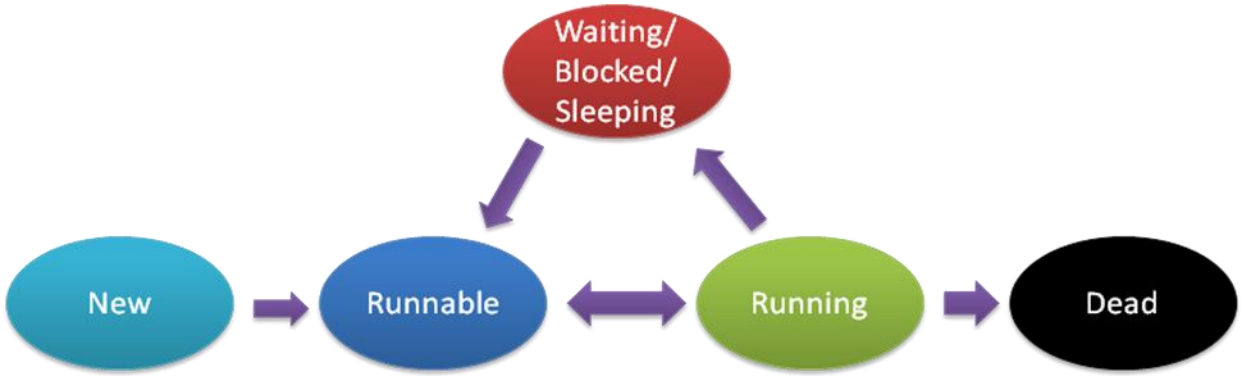
Output:

```
Main:0: Thread name is main
Main:1: Thread name is main
Run:0: Inside MyThread Run method
Run:1: MyThread name is main
Main:2: Thread name is main
Main:3: Thread name is main
Run:0: Inside MyThread Run method
Run:1: MyThread name is Java-Group
```

Note:

- **A thread is done being a thread when its target run() is completes**
- **Once a thread has been started, it can never be started again**
  If we try, we will get a RuntimeException - IllegalThreadStateException

THREADS STATE AND TRANSITION



NEW

This is the state the thread is in after the Thread instance has been created, but the start() method has not been invoked on the thread. It is a live Thread object, but not yet a thread of execution. At this point, the thread is considered not alive

RUNNABLE

This is the state a thread is in when it's eligible to run, but the scheduler has not selected it to be the running thread. A thread first enters the runnable state when the start() method is invoked, but a thread can also return to the runnable state after either running or coming back from a blocked, waiting, or sleeping state. When the thread is in the runnable state, it is considered alive.

RUNNING

This is the state a thread is in when the thread scheduler selects it (from the runnable pool) to be the currently executing process. A thread can transition out of a running state for several reasons, including because "the thread scheduler felt like it."

WAITING/BLOCKED/SLEEPING

This is the state a thread is in when it's not eligible to run. Okay, so this is really three states combined into one, but they all have one thing in common: **the thread is still alive, but is currently not eligible to run**. In other words, it is not runnable, but it might return to a runnable state later if a particular event occurs.

DEAD

A thread is considered dead when its **run() method completes**. It may still be a viable Thread object, but it is no longer a separate thread of execution. Once a thread is dead, it can never be brought back to life!

Reference:

Garbage Collection:
http://www.brpreiss.com/books/opus5/html/page414.html#SECTION001400000000000000000

Design pattern:
http://www.oodesign.com/

How HashMap works

http://coding-geek.com/how-does-a-hashmap-work-in-java/
http://www.java2blog.com/2014/02/how-hashmap-works-in-java.html