**JAVA**

**Java Features:**

\*\*\*\*\*\*\*\*\*\*\*\*\*

1)Simple programming

2)Object-Oriented programming language

3)Portable

4)Platform independent

5)Secured

6)Robust

7)Architecture neutral

8)Interpreted

9)High Performance

10)Multithreaded

11)Distributed

12)Dynamic



1)Simple Programming:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun, Java language is a simple programming language because:

* Java syntax is based on C++ (so easier for programmers to learn it after C++).
* Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
* There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

2)Objected-Oriented programming language:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Java is an [object-oriented](https://www.javatpoint.com/java-oops-concepts) programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behaviour.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Basic concepts of OOPs are:

1. [Object](https://www.javatpoint.com/object-and-class-in-java)
2. Class
3. [Inheritance](https://www.javatpoint.com/inheritance-in-java)
4. [Polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java)
5. [Abstraction](https://www.javatpoint.com/abstract-class-in-java)
6. [Encapsulation](https://www.javatpoint.com/encapsulation)

3) Platform independent:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

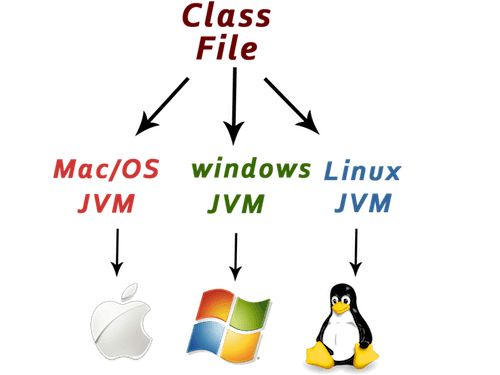
Java is platform independent because it is different from other languages like [C](https://www.javatpoint.com/c-programming-language-tutorial), [C++](https://www.javatpoint.com/cpp-tutorial), etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.

There are two types of platforms software-based and hardware-based. Java provides a software-based platform.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:

1. Runtime Environment
2. API(Application Programming Interface)

Java code can be run on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into byte code. This byte code is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere(WORA).



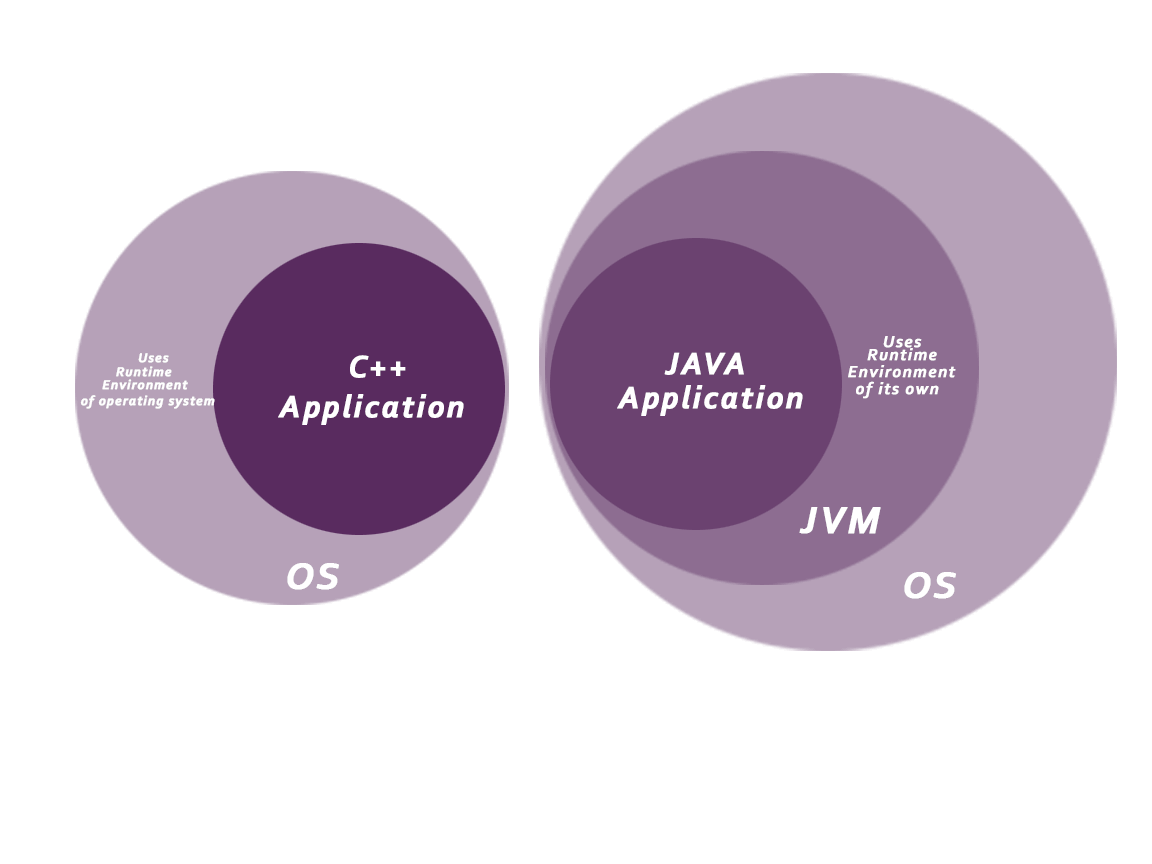
4)Secured:

\*\*\*\*\*\*\*\*\*

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

* No explicit pointer
* Java Programs run inside a virtual machine sandbox
* Class loader: Class loader in Java is a part of the Java Runtime Environment(JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
* Byte code Verifier: It checks the code fragments for illegal code that can violate access right to objects.
* Security Manager: It determines what resources a class can access such as reading and writing to the local disk.

Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.



### 5) Robust:

### \*\*\*\*\*\*\*\*\*

Robust simply means strong. Java is robust because:

* It uses strong memory management.
* There is a lack of pointers that avoids security problems.
* There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
* There are exception handling and the type checking mechanism in Java. All these points make Java robust.

6)Architecture-neutral:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

7)Portable:

\*\*\*\*\*\*\*\*\*

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

8)High-Performance:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

9)Distributed:

\*\*\*\*\*\*\*\*\*\*\*

Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

10)Multi-Threaded:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

11)Dynamic:

\*\*\*\*\*\*\*\*\*\*\*

Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

**Keyword:**

\*\*\*\*\*\*\*\*\*

Keyword is a predefined word. All keyword in the lower case.

**Identifiers:**

\*\*\*\*\*\*\*\*\*\*\*

The name is given by developer is called as identifier.

**Literals:**

\*\*\*\*\*\*\*\*

Value assigned to the variable is called literal.

**Variable:**

\*\*\*\*\*\*\*\*\*

Variable is a name of memory location. We have 3 types of variables

1)Local Variables

2)Instance variables

3)Static variables

**1)Local Variables:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪Local variable is declared inside the method, constructor and block.

🡪Local variable will be created when the method, constructor (or) block will be executed.

🡪In Local variable, variable is destroy when method execution.

🡪Local variable doesn’t have default value. So it is must be initialized before using it.

🡪Scope the local variable is inside method only.

🡪Access modifier can be used with local variable.

**2)Instance Variables:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪Instance variables declared inside the class but outside the method, constructor and block.

🡪When the space object except heap memory a space

🡪Instance variables are created when the object is created with the help of new keyword and it will destroy when the object destroy.

🡪For the instance variable access modifiers are a lot.

🡪Scope of instance variable inside method, constructor and block in the current class.

🡪For instance variable initialization is not mandatory because at the time of compilation compiler will put default for instance variable.

Note🡺method always preferences to local variables.

**Data Types:**

\*\*\*\*\*\*\*\*\*\*\*

Data type is diffenciate the value of the variable. By seeing the data type the memory will be allocated inside ram (Or) heap with the help of JVM.

Two types of Data types

1) Primitive Data type

2) User-Defined data types (or)object data types.

**Primitive Data Types:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Byte------🡪1 byte

Short-----🡪2 bytes

Numeric Data types

Int--------🡪4 bytes

Long------🡪8 bytes

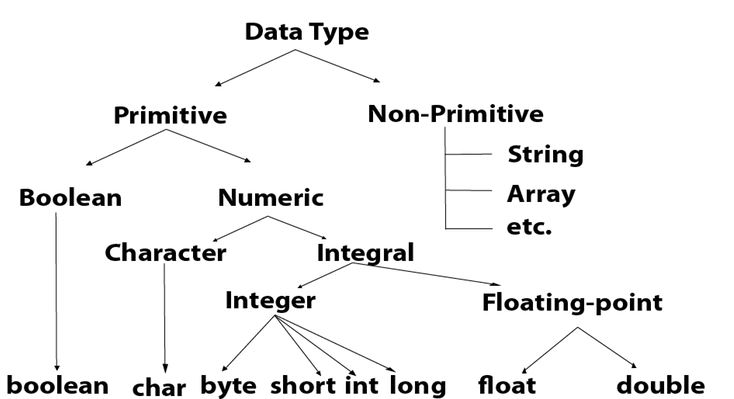
Float------🡪4 bytes

Decimal Data types

Double----🡪8 bytes

Char------🡪2 bytes

Boolean--🡪1 bit



**Operators:**

\*\*\*\*\*\*\*\*\*\*

Operator is a like a symbol to perform some operations.

**Types of Operators:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1) Arithmetic Operators 2) Logical Operators

3) Assignment Operators 4) Unary Operators

5) Concatenation Operators 6) Relational Operators

7) Bitwise Operators

**Conditional Statements:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1) If

2) Else

3) Else-If

4) Switch

1)If

\*\*\*\*

It can expects only Boolean.

2)Else:

\*\*\*\*\*\*

It will be executed when the condition will be false.

3)Else-If

\*\*\*\*\*\*\*\*

It executes one condition from multiple statements.

1)If 2)Else 3)Else-If

\*\*\*\* \*\*\*\*\*\* \*\*\*\*\*\*\*\*

Others

Else

IF

COND

**Switch:**

\*\*\*\*\*\*\*

🡪Java switch statement executes one statement from multiple conditions it is like as if-else-if ladder statement.

🡪Switch can expect only byte, short, int, char and a String an argument.

🡪Float, double and long not allowed for switch argument because they have more possibilities.

🡪If the case value is matched with expressions than the particular case will be executed.

🡪If there are no cases matched then the default case will be executed.

🡪The class value (or) label must be unique, duplicate case not allowed.

🡪The case label must be constant expressions (or) value if we providing variable as a case label then compile rise an exception.

🡪In switch Statement, case optional.

🡪In switch statement case and default both are optional.

🡪Inside the switch statement independent statements are not allowed.

\*🡪Inside the switch statement break is optional. If you are not providing break statement in that statement from the match case value upto break a statement will be executed.

🡪If no break is available upto the end of the switch statement till default case will be executed.

🡪Default statement we can write anywhere (starting, middle, ending).

🡪Switch argument data type and case label value must be same type.

**Loop (or) Iterative Statements:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

If we want to execute group of statements (or) more no of time than we are going for looping statement.

1. For loop

2) While loop

3) do-while loop

|  |  |  |  |
| --- | --- | --- | --- |
| Comparison | For Loop | While loop | Do-while loop |
| Introduction | The Java for loop is a control flow statement that iterates a part of the programs multiple times. | The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition. | The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition. |
| When to use | If the number of iteration is fixed, it is recommended to use for loop. | If the number of iteration is not fixed, it is recommended to use while loop. | If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop. |
| Syntax | for(init;cond;incr/decr)  {  // code to be executed  } | while(cond)  {  //code to be executed  } | Do  {  //code to be executed  }  while(condition); |
| Example | //for loop  for(int i=1;i<=10;i++)  {  System.out.println(i);  } | //while loop  int i=1;  while(i<=10)  {  System.out.println(i);  i++;  } | //do-while loop  int i=1;  do  {  System.out.println(i);  i++;  }  while(i<=10); |
| Syntax for infinitive loop | for(;;){  //code to be executed  } | while(true){  //code to be executed  } | do{  //code to be executed  }  while(true); |

Break

\*\*\*\*\*

**Continue:**

\*\*\*\*\*\*\*\*\*

Java continue Statement

It will continue the current flow of program it is escape remaining code at the specified condition.

🡪In case of inner loop it will continue inner state only.

**Access Modifiers:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Using access modifiers we can provide the accessibility level of the class, method and data member.

1) Public

2) Default

Outer class

3) Final

Inner Class

4) Abstract

5) Strictfp

6) Private

7) Protected

8) Static

9) Synchronize

10) Native

11) Transient

12) Volatile

Public:

\*\*\*\*\*\*

The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Default:

\*\*\*\*\*\*\*

The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

Private:

\*\*\*\*\*\*\*

The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

Protected:

\*\*\*\*\*\*\*\*\*

The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

Final:

\*\*\*\*\*

Java final keyword is used to indicate that a variable holds a constant value. It is applied with a variable. It is used to restrict the user.

Strictfp:

\*\*\*\*\*\*\*

Java strictfp is used to restrict the floating-point calculations to ensure portability.

Abstract:

\*\*\*\*\*\*\*\*

Java abstract keyword is used to declare abstract class. Abstract class can provide the implementation of interface. It can have abstract and non-abstract methods.

Static:

\*\*\*\*\*\*

Java static keyword is used to indicate that a variable or method is a class method. The static keyword in Java is used for memory management mainly.

Synchronize:

\*\*\*\*\*\*\*\*\*\*\*

Java synchronized keyword is used to specify the critical sections or methods in multithreaded code.

Native:

\*\*\*\*\*\*

Java native keyword is used to specify that a method is implemented in native code using JNI (Java Native Interface).

Transient:

\*\*\*\*\*\*\*\*\*

Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.

Volatile:

\*\*\*\*\*\*\*

Java volatile keyword is used to indicate that a variable may change asynchronously.

**Q) Difference between Access Specifier and Access Modifier?**

In c and c++ public, private, protected and default are considered as access specifier and rest are access modifier. But in java all are considered as access modifier.

**OOPS Concepts:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

It is a methodology to design a program using classes and objects.

1) Class

2) Object

3) Inheritance

4) Polymorphism

5) Abstraction

6) Interface

7) Encapsulation

**Class:**

\*\*\*\*\*\*

🡪Class is a blue print from which individual object can be created.

🡪Class is a template (or) blue print (or) proto type of the object.

🡪Class is a logical entity.

🡪Class contains data members, methods and constructors.

**Syntax of Class:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[Access modifier] class ClassName extends AbstractClass implements Interface

{

Datamemebers,

Methods,

Constructors,

Blocks

}

**Object:**

\*\*\*\*\*\*\*

In real world objects they two characteristics

1. State 2) Behaviour
2. What is possible state.
3. What is behaviour can this object perform.
4. Object is an instance of the class.
5. Object is a physical entity.
6. Object contain in member function.

**Object Syntax:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Employee emp=new Employee();

Constructor

ClassName

Reference variable

**Method:**

\*\*\*\*\*\*\*

🡪Method is describe behaviour of an object.

🡪A set of Statements we are making a group to perform some a specific task.

🡪Method created ones and we can execute many times.

**Syntax of Method:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[Access modifiers] return type method name(parameter List) throws Exceptions

Optional

Optional

{

--------------------------

----------------------------

-------------------------

}

**Return Type:**

\*\*\*\*\*\*\*\*\*\*\*\*

🡪Return type is not mandatory for void.

🡪Void cannot accept any return values.

🡪Return statements mandatory all these method expect void method.

🡪The method return type and return value should be same else will get compile time error.

🡪Return statement should declared last in method.

**Arguments & Parameters:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪Parameters are variable declared in the method signature.

🡪Parameter is used to receive input value from calling method.

🡪Arguments are variable and it is used to pass in the method call.

🡪Scopes of the argument are the parameters with in the current method.

**Casting:**

\*\*\*\*\*\*\*\*

Converting one data type to another data type is called as casting.

Casting

Derived casting/user defined casting

Primitive casting

Down Casting

Up Casting

Narrowing

Widening

**Primitive casting:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Converting one primitive data type to another primitive data type is called as casting.

Datatype2 var2=(Datatype2) var1;

**Widening:**

\*\*\*\*\*\*\*\*\*

Converting a smaller data type to big data type is called widening.

**Note:**

\*\*\*\*\*

🡪For widening no need to follow any syntax at the time of compilation compiler will provide default syntax for widening.

🡪Widening is implicit.

**Narrowing:**

\*\*\*\*\*\*\*\*\*\*

Converting the bigger data type to smaller data type is called narrowing.

**Note:**

\*\*\*\*\*

Narrowing is explicit so we must follow the syntax otherwise will compilation error.

Implicit (or) widening

Bigger data type

Smaller data type

Explicit (or) Narrowing

A boolean value cannot be assigned to any other data type expect boolean, all the remaining seven data types can be assigned to other data type either implicitly (or) explicitly but Boolean cannot.

🡪We can say Boolean is a in compatible type for conversion.

🡪Maximum we can assign another a Boolean value.

Boolean b=true;

Int i=(int)b;

System.out.println(i); //incompatible type.

**Method Overloading:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪Create the multiple method with same name but different argument list is known as Method Overloading.

🡪Different argument list means argument can be differ in number (or) which type.

🡪If we are passing the matching type argument then type casting will not perform.

🡪If we are not passing matching type argument will get compile time error.

🡪When the functionality is same then we should go for overloading.

🡪User can be remembered one method for one functionality.

🡪Overloading is achieving the compile time polymorphism.

🡪We can overload main method i.e public static void main().

**EX:**

\*\*\*

🡪Print and println

🡪driver.switchTo().Frame(String name or Id);

🡪driver.switchTo().Frame(Int Index);

🡪driver.switchTo().Frame(WebElement w1);

**Constructor:**

\*\*\*\*\*\*\*\*\*\*\*

🡪It is used to initialize our instance variable and instance member.

🡪It is a special type of method.

🡪Constructor is used to create object.

🡪The role of the constructor in object creation is to provide initial values inside the object.

🡪In java applications, constructor are executed exactly at the time of creating objects, not before create objects and not after creating objects.

🡪To provide initialization for class level variables mainly instance variable.

🡪Constructor must have same name of the respective class.

🡪Constructors are not having return types.

🡪Constructors are not allowing access modifiers like static, final,……

🡪Constructors are able to allow access modifiers like public. Private, protected and Default.

🡪Constructors are allowing ‘throws’ keyword in its syntax to bypass exception from present constructor to the caller.

🡪If our class doesn’t contains any constructor then at the time compilation compiler will put default constructor for the class.

Syntax:

\*\*\*\*\*\*\*

[AccessModifier] Class\_name([Param\_List]) [throws Exception\_List]

Constructors are two types

1)Default constructor 2)Parameterized constructor

**1)Default constructor:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪Constructor without parameter is called default constructor.

🡪If your class contains any constructor whether it is default (or) parameterized, then the compiler will not put any constructor in that class.

**2)Parameterized constructor:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Constructor with parameters is called parameterized constructor.

**This Keyword:**

\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪This keyword is used to refer the current class instance members (or) instance variables.

🡪If the local variable and instance variable name is same then we are using this keyword to different local variable and instance variable.

🡪This keyword is used to current class method also.

🡪If you are not using this keyword with method at the time of compilation compiler will default this keyword.

**Constructor Overloading:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪Creating the multiple constructors in a same class but different argument list.

🡪Different argument list means argument can be different will length (or) type.

🡪If you are not passing matching type argument then it will compile time error.

**Call to this:**

\*\*\*\*\*\*\*\*\*\*\*

🡪Call to this is used to call the current class constructor.

🡪Call to this will connect to the constructor body based on the parameter.

🡪Call to this is used to call one constructor inside the another constructor without creating the object.

🡪Call to this must be first Statement in constructor.

**Syntax:**

\*\*\*\*\*\*\*

This(parameter);

Here parameter is optional.

**Difference between Method and Constructor?**

|  |  |
| --- | --- |
| Method | Constructor |
| 🡪Method has return type. | 🡪Constructor does not have return type. |
| 🡪Method name can any valid name. | 🡪Constructor name can be only class name. |

🡪Method we can call externally (or) anywhere but constructor we can only at the time of object creation.

**Static Keyword:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪It is a keyword.

🡪It is non-access modifier which is used to save our memory.

🡪A static member are loaded in a static segment at the time class loading.

🡪Class will load 1st time when it is created.

🡪Static member we can access inside the static method (or) non-Static method directly (or) with the class name.

🡪Static member we can access inside the non-Static method (or) static method by using class name in a different class.

**Syntax:**

\*\*\*\*\*\*\*

className.Staticmember(variable and method);

**For non-Static syntax:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ObjectRef.Nonstaticmember(variable and method)

If you call non static member inside a static method mandatory we need to create object and we can call will object reference.

**Note:**

\*\*\*\*\*

A Static member we can call with object reference also because all the static member will be the common properties of all these object.

**Difference between a static and non-Static?**

|  |  |
| --- | --- |
| Static | Non-Static |
| 🡪A static member will be loaded into static segment (or)static pool at time of class loading. | 🡪Non-Static will be created inside the heap at the time of object creation with the help of new keyword. |
| 🡪For Static members we have only one copy. | 🡪For non-static member we have multiple copy(how many objects we are creating that many copy will be available). |
| 🡪A static member we can call with the class name as well as with object reference. | 🡪Non-Static members we can call only with object reference. |
| 🡪A static members we can call before creating the object. | 🡪Non-Static members we can call only after creating the object. |

**Inheritance:**

\*\*\*\*\*\*\*\*\*\*\*

Getting properties and behaviours from one class to another class is called as inheritance.

🡪The main purpose of inheritance is code re-usability.

🡪Inheritance is also known as Is-A relationship i.e two-classes are belonging to same hirechry.

🡪In inheritance which is giving the properties is called as parent class (or) super class.

🡪The class which is getting the properties is called as child class (or) sub class.

🡪To create a “Is-A” relationship b/w to classes we can use “Extends” Keyword.

🡪Private members cannot be inherited.

**Types of Inheritances:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1)Single Level Inheritance:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In Single level inheritance only one sub class will extend one super class.

A

Extends

B

2)Multilevel Inheritance:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In multilevel Inheritance one sub class is accessing the properties of super class and that super class is accessing the properties of their own class.

A

Extends

B

Extends

C

3)Multiple Inheritance:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

One sub class is getting the properties and behaviours from more than one super class is known as Multiple inheritance.

Multiple inheritance is not possible in java.

B

A

C

4)Hierarchical Inheritance:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪Multiple sub class will get the properties from one single super class.

🡪Multi sub classes is extending one super class is called Hierarchical Inheritance.

A

E

D

C

B

5) Hybrid Inheritance:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Combinations of any two inheritances is called as Hybrid Inheritance.

O

A

E

C

B

D

**Super Keyword:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪Super keyword is used to refer to immediate parent class instance member.

🡪Super keyword is used to refer to immediate parent class method.

**Call to Super:**

\*\*\*\*\*\*\*\*\*\*\*\*\*

It is used to refer immediate parent class constructor.

**Syntax:**

\*\*\*\*\*\*\*

Super(parameter)

Here parameter is a optional.

🡪By default call to super will be available for every constructor.

🡪If you are not providing call to super then at the time of compilation compiler will put default call to super.

🡪Call to super will invoke immediate parent class constructor based on the parameter list.

🡪Call to super will be the 1st statement inside the constructor. So while using call to this we cannot use call to super.

**Aggregation (Has-A Relationship):**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪If a class contains reference of another class it is called Aggregation (or) Has-A Relationship.

🡪When there is no relation b/w two classes then we have to go for Has-A to use one class properties to another class properties.

**Method Overriding:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Changing of implementation of inherited method (parent class (or) super class) in a sub class based on sub class requirement without changing the signature of method.

(or)

If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

**Rules for overriding:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪Inheritance is mandatory.

🡪In sub class method name, argument list, return type should be same as super class method.

🡪In sub class access modifier can be change.

🡪We can increase the visibility level for overridden method but we cannot reduce it.

🡪Private method cannot be overridden because it cannot be inherited sub class.

|  |  |
| --- | --- |
| Super class | Sub class |
| Private | X |
| Default | Default, protected, public |
| Protected | Protected, public |
| Public | public |

Private

Default

Protected

Public

**Up Casting:**

\*\*\*\*\*\*\*\*\*\*

🡪Creating the object of sub class and assign to the super type reference variable is known as up casting.

🡪If we are doing up casting by using super type ref variable we can call only inherited method, we cannot call to sub class specific member.

**Down Casting:**

\*\*\*\*\*\*\*\*\*\*\*\*\*

Converting the super type to sub class type but super class type should be updated.

**Method Hiding:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪A Static method can be overloaded but cannot be overridden, if you are overriding a static method inside a sub class then sub class implementation will hide super class a static method implementation.

🡪Creating the static method in a sub class which is adjently as super class a static method signature is known as method hiding.

**Final Keyword:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Final keyword is a keyword which is used with class variable and method.

Variable:

\*\*\*\*\*\*\*\*

If you are defined any variable as a final that variable cannot be change. Final variable constant in java.

Final method:

\*\*\*\*\*\*\*\*\*\*\*

🡪If you are defined any method as a final that the method cannot be overridden.

🡪Final method can be inherited but cannot be overridden.

**Difference B/W Overloading and Overriding?**

|  |  |
| --- | --- |
| Overloading | Overriding |
| 1) Whenever same method or Constructor is existing multiple times within a class either with different number of parameter or different type of parameter or with different order of parameter is known as Overloading. | 1)Whenever same method name is existing multiple time in both base and derived class with same number of parameter or same type of parameter or same order of parameters is known as Overriding. |
| 2) Arguments of method must be different at least arguments. | 2) Argument of method must be same including order. |
| 3) Method Signature must be different. | 3) Method Signature must be same. |
| 4) Private, Static and Final methods can be overloaded. | 4) Private, Static and Final methods cannot be override. |
| 5) Access modifiers point of view no restriction. | 5) Access modifiers point of view not reduced scope of Access modifiers but increased. |
| 6) Also known as compile time polymorphism or static polymorphism or early binding. | 6) Also known as run time polymorphism or dynamic polymorphism or late binding. |
| 7) Overloading can be exhibited both are method and constructor level. | 7) Overriding can be exhibited only at method label. |
| 8) The scope of overloading is within the class. | 8) The scope of Overriding is base class and derived class. |
| 9) Overloading can be done at both static and non-static methods. | 9) Overriding can be done only at non-static method. |
| 10) It is used to increase the readability of the program. | 10) It is used to provide the specific implementation of the method that is already provided by its super class. |
| 11) It is performed within class. | 11) It occurs in two classes that have IS-A(inheritance) relationship. |
| 12) It is the example of compile time polymorphism. | 12) It is the example of run time polymorphism. |
| 13) For Overloading methods return type may or may not be same. | 13) For Overriding method return type should be same. |

**Binding:**

\*\*\*\*\*\*\*\*

Connecting the method call to method body is called as Binding.

**Two Types of Binding:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1)Static Binding (or) Compile time Binding (or) Early Binding

2)Dynamic Binding (or) Late Binding (or) Run time Binding

**Static Binding:**

\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪When type of object is determined at the time of compilation is known as ~. (or)

When the method call to method body at the time of compilation is known as Static binding.

🡪For a static binding inheritance is not required.

🡪For private, final (or) static method in a class there will be static binding.

**Dynamic Binding:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪When the type of object is determined at runtime is known as Dynamic binding.

🡪For Dynamic binding inheritance is required.

**Polymorphism:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

One object will perform single operation in different ways is called ~ (or) one object showing different behaviours in it’s life cycle is called polymorphism.

1)Compile time polymorphism 2)Dynamic Polymorphism

1)Compile time polymorphism:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Call to overloaded method is resolved at compile time based on argument list.

2)Dynamic polymorphism:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Call to overridden method is resolved at runtime based on the object creation.

**Abstraction:**

\*\*\*\*\*\*\*\*\*\*\*

Hiding the implementations by providing common interface.

**Concrete method/fully constructed method:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Any method which contains both signature and body is known as concrete method.

**Abstract method:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Any method which has only signature, then don’t have any body is called as abstract method.

🡪Abstract is a keyword we are using to define abstract method.

🡪Abstract method should be declared with abstract keyword and it’s should be end with semicolon.

**Concrete Class:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

If any class contains only concrete method it is called as concrete method.

**Abstract class:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪Abstract class we are defining with abstract keyword.

🡪In abstract class abstract method is optional. In abstract it can have abstract method (or) end concrete method also.

🡪If a class contains at least one abstract method then that class should be declared as abstract class else will get compile time error.

🡪If a class declared as abstract then we cannot create object for abstract class. But we can create reference of abstract class.

🡪When a class extends abstract class all this abstract method of super class should be overridden in sub class otherwise sub class will be abstract.

🡪Abstract class can have only concrete method.

🡪We cannot create object of abstract class.

🡪If any class contains abstract method then that class must be defined as abstract.

🡪The abstract method is super class type can increase the visibility inside the sub class.

🡪Private, final and Static cannot be abstract.

Abstract final,

Invalid combinations

Abstract Static,

Abstract private

🡪All the abstract class will tell to sub class forcefully to provide implementation of abstract method otherwise we have to may sub class as a abstract.

🡪If your class contains only static method better makes that class as abstract class.

**Coupling:**

\*\*\*\*\*\*\*\*\*

There are two types of couplings 1) loose coupling

2) Tight coupling

**Loose Coupling:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

When the two API (or) two programmes are interconnected with such a way that any enhance in one API does not effect of another API.

**Tight Coupling:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

When two API (or) two programmes are interconnected such a way that any enhancement (or) modification in one API can effect to the other API is called as Tight coupling.

**Interface:**

\*\*\*\*\*\*\*\*\*

🡪Interface is a reference for any class. It is a bridge b/w (or) it is used to as bridge b/w two programs.

🡪In interface all these method are public and abstract(by default).

🡪In interface all the variables are by default public static final.

🡪To define interface we are using keyword interface.

🡪We use implements keyword to implement interface to class.

🡪For interface variable should be initialized otherwise it will throw compile time error.

🡪In interface we cannot create object. But we can create reference.

🡪We don’t have any constructor in interface.

🡪Java compiler add public and abstract keyword before all the methods if we don’t provide.

🡪Public static final keyword before all the data member.

🡪When the class will implements interface all the interface method should be overridden inside the sub class otherwise sub class becomes abstract.

🡪While implementing interface method visibility access should be public because in interface by default all the methods are public.

🡪Advance of interface over abstract class.

🡪Using interface we can achieve multiple inheritance.

🡪Using interface we can achieve 100% abstraction.

🡪Using interface we can achieve loose coupling.

**Difference B/W Interface and Abstract class?**

|  |  |
| --- | --- |
| Interface | Abstract Class |
| 1) In Interface by default all the methods are public and abstract. | 1) In abstract we can have both abstract method as well as concrete method. |
| 2) In interface by default all the variables are public static final. | 2) In abstract class variables can be static, non-static, final..etc. |
| 3) In interface initialization of variable is mandatory. | 3) In abstract class initialization of variable is not mandatory. |
| 4) We cannot have constructor in interface. | 4) We can have constructor in abstract class. |
| 5) A class can implement multiple inheritance. | 5) A abstract class can extend only one class. |

**Instance of operator:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪A instance of operator is use to verify whether the object is an instance of specific type(class, sub class (or) interface).

🡪If you are creating object of sub class it will be type of super class (or) super type interface also, i.e using the sub class reference variable we can call method of super class.

Ex:

\*\*\*

Class A

{

Void m1()

{

System.out.println(“M1 of A”);

}

}

Class B extends A

{

Void m1()

{

System.out.println(“M1 of B”);

}

}

Class Run

{

Public static void main(String[] args)

{

A a1=new A();

Boolean b1=a1 instance of A;

System.out.println(b1); // true

A a2=new A();

Boolean b2=a2 instance of B();

System.out.println(b2); //true

}

}

**Encapsulation:**

\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪It is a technique of making the filled private in that filled we can access through some public method. That is called as getter and setter method.

🡪Public method which is used to set the private field is called as setter.

🡪Public method which is used to read (or) fetch the private field is called as getter.

🡪Encapsulation is also define as data hiding because variables are private.

**Java.Lang.Packages:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

For writing any java program the most commonly required classes and interfaces are encapsulated in the separate package is called as Java.lang.packages.

1) Object

2) String

3) String Builder

4) String Buffer

5) Wrapper class

6) Boxing and Un Boxing

7) Utility Methods

8) Array

**Object class:**

\*\*\*\*\*\*\*\*\*\*\*\*

🡪For any object whether it is pre-defined (or) user defined the most commonly required method are encapsulated into separate class which is called as object class.

🡪A object X has a route (or) parent (or) super class for all this classes.

🡪By default it method’s will be available in every classes.

Note:

\*\*\*\*\*

🡪If our class does not extend any other class then it is directly inheriting object class.

Class A

Object

{

}

A

Class B extends A

{

B

}

🡪If our class extends any other class then it is indirectly inheriting object class.

**Object Class Methods:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* Public string toString()
* Public int hashCode()
* Public Boolean equals(object o1)
* Public final class getClass()
* Protected final objectClone()
* Protected void finalize()
* Public final void wait()
* Public final native wait(long m1)
* Public final void wait(long m1,int m1)
* Public final void notify()
* Public final void notifyAll()

**toString():**

\*\*\*\*\*\*\*\*\*\*

**Syntax:**

\*\*\*\*\*\*\*

Public string toString()

{

---------

---------

return “ “

}

This method is used to get the object string representation of an object. Whenever we trying to print any object reference internally to String method executed.

🡪If our class does not contains toString() method then object class toString method executed.

(or)

🡪toString() returns string representation of the object.

🡪If you want print any object, java compiler internally invokes toString() method on the object.

🡪So overriding the toString() method, returns of the desired O/P.

🡪In a String, string builder, string buffer, wrapper classes and all the collection classes toString method is overridden for meaningful representation.

**Public native int hashCode():**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪For every object JVM will generate unique number which is called as hashcode.

🡪JVM uses hash code while searching the object into hashing like hash set, hash table and hash map etc.

🡪if you don’t override hash code method then object class hash code method will be executed, which generate hash code based on the address of the object.

Public int hashCode()

{

Return id;

}

**Note:**

\*\*\*\*\*

🡪If you are giving to the object class to a string method internally called hashCode() method.

🡪But if we are overriding to a string method then it may not call hashCode method().

**Equalls() method:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪.Equal(); used  to check reference comparison(address) for user created objects only.

🡪.Equal(); used  to check content comparison for user created objects when .equal() is overridden.

And remember this

🡪.Equals () by default overridden in STRING, ARRAY, COLLECTIONS, MAPS…

🡪So in case of this equals method used for content comparison only.

==

\*\*\*\*

🡪“==”operator  is used to check content comparison in case of Primitives.

(Byte, short, int, long, float, double, boolean are primitives)

🡪“==”operator is used to check reference comparison (address) in case of objects.

(User defined objects, String, array, collections are objects)

🡪“==” operator compares the two objects on their physical address. That means if two references are pointing to same object in the memory, then comparing those two references using “==” operator will return true. For example, if s1 and s2 are two references pointing to same object in the memory, then invoking s1 == s2 will return true. This type of comparison is called “Shallow Comparison”.

🡪equals() method, if not overrided, will perform same comparison as “==” operator does i.e comparing the objects on their physical address. So, it is always recommended that you should override equals() method in your class so that it provides field by field comparison of two objects. This type of comparison is called “Deep Comparison”.

🡪 In java.lang.String class, equals() method is overrided to provide the comparison of two string objects based on their contents. That means, any two string objects having same content will be equal according to equals() method. For example, if s1 and s2 are two string objects having the same content, then invoking s1.equals(s2) will return true.

**Examples of Equalls() and ==:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪Let’s apply these three methods on string objects and try to analyse their output

Define two string objects like below,

String s1 = "JAVA"

String s2 = "JAVA";

Now apply above methods on these two objects.

s1 == s2 —> will return true as both are pointing to same object in the constant pool.

s1.equals(s2) —> will also return true as both are referring to same object.

🡪This type of comparison is straight forward. There is no speculation about this comparison. Let’s define the string objects like below,

String s1 = new String("JAVA");

String s2 = new String("JAVA");

s1 == s2 —> will return false because s1 and s2 are referring to two different objects in the memory.

s1.equals(s2) —> will return true as both the objects have same content.

🡪Comparing the string objects defined like below will also give same result as the above.

String s1 = "JAVA";

String s2 = new String("JAVA");

s1 == s2 —> will return false because s1 and s2 are referring to two different objects in the memory.

s1.equals(s2) —> will return true as both the objects have same content.

🡪Now, you may conclude that If there is a requirement of comparing two string objects on their physical address, then use “==” operator and if there is a requirement of comparing two string objects on their contents, then use equals() method or hashCode() method.

Hold on…. Before jumping onto conclusion, compare these two string objects.

String s1 = "0-42L";

String s2 = "0-43-";

s1 == s2 —> will return false as s1 and s2 are referring to two different objects in the memory. (Expected…)

s1.equals(s2) —> It will also return false as both the objects have different content. (Expected…)

**CLONE():**

\*\*\*\*\*\*\*\*\*

🡪The purpose of creating urgently duplicate object is called as clone().

🡪The main objective of cloning is backup purpose.

🡪We can perform cloning by using clone() which present in object class.

🡪We can perform cloning only for clonable object.

🡪An object to be clonable if only if the corresponding class implement clonable interfaces.

**getClass():**

\*\*\*\*\*\*\*\*\*\*\*

🡪This method returns the run time class definition of an object.

Ex:

\*\*\*

Class A

{

-----

-----

}

Class Run

{

Public static void main(String[] args)

{

A a=new A();

Class c1=a.getClass();

String classname=c1.getName();

Sytem.out.println(classname);

}

}

**String:**

\*\*\*\*\*\*

🡪A String is a combination of multiple characters.

🡪String is final class present in java.lang.packages.

🡪There are two-ways we can create object (or) for a string.

i)A string literal (or) using string literal

ii) By using new keyword

SCP

SCP

**i) A String literal:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

S1

String s1=”sai”;

String s2=”kiran”;

S2

kiran

Sai

String s3=”Sai”;

S3

🡪When we are creating the object a string literal an object will be created inside the string constant pool.

🡪Object creation in a SCP is always optional before an object in a SCP.

🡪1st JVM will check is there any object is present with the same content (or) not.

🡪If it is present then JVM will reuse the existing object instead of creating new object.

🡪If it is not available then will create new object.

**ii)By using new keyword:**

SCP

heap

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1)String s1=new String(“sai”);

2)String s2=”sai”;

Sai

Sai

3)String s3=new String(“sai”);

Sai

🡪In the above example two objects created.

i)The one object will created inside the heap

because of new keyword and another object

will be created in SCP because string literal

with the same content and reference will be

pointed to heap memory.

🡪Garbage collector cannot access a string constant pool area hence even through if any object doesn’t have reference still that object is not eligible for garbage collector.

🡪All the SCP object will be destroy at the time of JVM shut down.

🡪Inside heap if any object doesn’t reference then that object will be eligible for GC.

🡪Once we are creating a string object we cannot perform any changes in the existing object.

🡪If we are trying to change then with those change a new object will be created with these changes.

🡪This behaviour is called as immutable of string object.

**Q)Why String is immutable?**

🡪One object can be referred by more than one reference variable.

🡪If any one reference variable wants to change the content of object then it will effect to other reference variables also.

🡪Hence String is immutable.

**String Buffer:**

\*\*\*\*\*\*\*\*\*\*\*\*

🡪If the content of a string object will change frequently then going for the string object then for every changes a new object will be created internally that will cause memory wastage.

🡪To remove the above problem we should go for string buffer concept.

🡪The main advantage of string buffer is all the required changes will be perform in the existing object itself.

🡪Creating an empty string buffer object with default capacity of 16.

🡪Once the string buffer object which is to maximum capacity then new a string buffer will be created with the new capacity.

**Note:**

\*\*\*\*\*

🡪Each method present in a string buffer is synchronized.

🡪Hence at a time only one thread is allow to operate on a string buffer object.

🡪It increase the waiting time thread hence it creates the perform issue.

🡪To overcome this we go for string builder.

**String Builder:**

\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪It is introduced in 1.5V

🡪Every method present in a string builder non-synchronized it means it is not thread safe.

🡪Hence at a time multiple thread(program) is allowed operate on string builder object, i.e it will increase the performance.

**Thread Safe** **Not Thread Safe**

P1

P1

Wait untill

String Buffer

P2

String Buffer

P2

P1 will execute

P3

Wait untill P1and p2 will execute

P1

**Difference B/W String Buffer and String Builder?**

|  |  |
| --- | --- |
| **String Builder** | **String Buffer** |
| 1) Method present in string builder is non-synchronized. | 1) Method present in string builder is synchronized. |
| 2) At a time multiple thread are allowed to operate simultaneously on string builder object. Hence it is not thread safe. | 2) At a time only one thread is allowed to operate on a string buffer object. Hence String Buffer is thread safe. |
| 3) Threads are not required to wait hence the performance will high. | 3) It increase the waiting time of the thread hence the performance is low. |
| 4) It is introduced in 1.5 V | 4) It is introduced in 1.0 V |

**Difference B/W String, String Buffer and String Builder?**

|  |  |  |
| --- | --- | --- |
| **String** | **String Buffer** | **String Builder** |
| 🡪String is immutable hence we cannot change the content of the object. | 🡪String Buffer mutable, we can change the content of the object. | 🡪String Buffer mutable, we can change the content of the object. |
| 🡪If the content is fixed and if won’t change frequently then we should go for String. | 🡪If the content change frequently and if we want to keep program thread safe then we go for String Buffer. | 🡪If the content change frequently but thread safe is not required. Then we go for String Builder. |

**Array:**

\*\*\*\*\*\*

🡪An array is an individual collection of fix the number of homogeneous data element.

🡪When we want to define the multiple value in a single variable of same type then we should go for array.

**Advantages of Array:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪We can represent the multiple value with a same name so that the readability of code will be improve.

**Dis-Advantages:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪It is fixed in size, that is once we create an array there is no chance of increasing (or) decreasing the array based on our requirement.

🡪Array can expect only homogenous element(same type).

**Syntax:**

\*\*\*\*\*\*\*

Datatype[] var1;

Datatype []var1;

Datatype var1 []

🡪Every array in java is an object hence we can create object by using new keyword.

Datatype var[]=new datatype[size];

**Ex:**

\*\*\*

* Int a[]=new int[5];
* Int a[]=new int[0];

Int a[]=new int[]; x

Int a[]=new int[-1];x

Int a[]=new int[10.02];x

**COLLECTIONS**



**Collections:**

\*\*\*\*\*\*\*\*\*\*\*

🡪The **Collection in Java** is a framework that provides architecture to store and manipulate the group of objects.

🡪Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

🡪Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

🡪If you are defining any collection we can increase or decrease the size middle of the program (or) code.

🡪In collection we can a store heterogeneous element.

🡪Every collection class is implemented based on some data stricter hence for every requirement we will have some inbuilt method to work on it.

**Advantages of Collection:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪If you are defining any collection we can increase or decrease the size middle of the program (or) code.

🡪In collection we can a store heterogeneous element.

🡪Every collection class is implemented based on some data structure.

🡪Hence for every requirement we will have some in built method to work on it.

**Difference B/W Array and Collection?**

|  |  |
| --- | --- |
| **Array** | **Collection** |
| 1) Array is fixed in size. | 1) Collection are grow able in a nature i.e increase (or) decrease size. |
| 2) Array can hold homogenous data. | 2) Collections can hold both homogenous and heterogeneous data. |
| 3) Array can hold both primitive and object type. | 3) Collection can hold only object type not primitive. |
| 4) Array concept is not implemented based on any data structure. | 4) For every collection class some data structure implemented based on requirement. |
| 5) Array is not recommended to use with respect to memory. | 5) With respect to memory collection are recommended to use. |
| 6)Array performance is high compare to collection. | 6) Collection is low compare to array. |

**What is collection?**

A Collection represents a single unit of objects, i.e., a group. (or)

If you want to represent a group of individual object in single entity then we should go for collection entity.

**What is a framework in java?**

* It provides readymade architecture.
* It represents a set of classes and interfaces.
* It is optional.

**What is a collection framework?**

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm

**(or)**

If defines several classes and interface which can be use to represent a group of object in a single entity.

Collection is a interface

Collections is a class

**Collections interface methods:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1) public abstract Boolean add(object o1);

2) public abstract Boolean addAll(collection c);

3) public abstract Boolean remove(object o1);

4) public abstract Boolean removeAll(collection c);

5) public abstract Boolean retainAll(collection c);

6) public abstract clear();

7) public abstract Boolean contains(object o1);

8) public abstract Boolean containsAll(collection c);

9) public abstract Boolean isEmpty();

10) public abstract intSize();

11) public abstract Iterator iterator();

**Note:**

\*\*\*\*\*

There is no concrete class which implements collection interface directly.

**List Interface:**

\*\*\*\*\*\*\*\*\*\*\*\*\*

It is a direct child Interface of collection, JDK 1.2v.

🡪It is index based, it able to arrange all the elements as per indexing.

🡪To allow duplicate elements, following insertion order, not follow sorting order, able to allow any no of null values, able to allow heterogeneous elements.

**When we go for list interface?**

🡪If you want to represent a group of individual object where the duplicates are allowed and insertion order should be preserved then we should go for list interface.

In list interface always a store a value with index basis.

**Specific methods of list interface:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Public void add(int index, Object o1);
2. Boolean addAll(int index, collection c);
3. Object get(int index);
4. Object remove(int index);
5. Public object set(int index, object o1)
6. int indexOf(object o1);
7. int lastIndexOf(object o1);
8. List Iterator list iterator();

**ArrayList:**

\*\*\*\*\*\*\*\*\*

🡪Jdk 1.2v, direct implementation class to List interface, index based, duplicate elements are allowed, follows insertion order, will not follow sorting order, allows heterogeneous elements, allows any no of null values.

🡪Internal data structure is “Resizable array”.

🡪ArrayList is best suitable for retrieving (or) fetching the data from the collection.

🡪intinal capacity is 10.

🡪All methods present in arrayList is non-synchronized.

🡪Its incremental capacity ratio is new capacity= (current capacity \* 3/2)+1.

🡪At a multiple threads are allowed to operate on ArrayList object simulatensiouly. (or) allows more than one thread to access data.

🡪Follows parallel execution, will reduce execution time, improve application performance, not give guarantee for data consistency, it is not thread safe, it is not legacy collection.

🡪ArrayList not best suitable for inserting (or) deleting the data from ArrayList in middle because lots of shifting requires to perform the operation.

ArrayList al=new ArrayList() 🡺it is a non-generic process

ArrayList<String or Object> al=new ArrayList<String or Object> 🡺 It is a generic process.

**Q) Difference between add() and set()?**

|  |  |
| --- | --- |
| **Add()** | **Set()** |
| Add() method is able to perform insert operation. | Set() method is able to perform replace operation |
| If any element is existed at the specific element then add() method will insert the specified new element at the specified index value and add() method will adjust the existed element to next index value. | If any element is existed at the specified index then set() method will remove the existed element and set() method will add the specified element to the specified index and set() method will return the remove element. |
| If no element is existed at the specified index then add() method add the specified element at the specified index. | If no element is existed at the specified index value then set() method will rise an exception like java.lang.indexOutOfBoundException. |

1)public void add(int index, object o1):

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Add element in b/w any element. (or) It able to add the specified element at the specified index value.

Ex:

\*\*\*

Al.add(3,”sai”);

2)Public Object set(int index, Object obj):

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Set method is used to replace the element present at the specified index with provided object and it will return the old object (or) it able to set the specified element at the specified index value.

EX:

\*\*\*

Al.set(2,”kiran”);

**LinkedList:**

\*\*\*\*\*\*\*\*\*\*

Jdk 1.2v, not legacy collection, direct implemented class to list interface, index based, allowed duplicate values, follows insertion order, not follow sorting order, allows null values, allows both heterogeneous and homogenous elements, not synchronized collection, no method is synchronized in Linked list, allows more than one thread to access data, will follow parallel execution, decrease execution time, improve application performance, not thread safe, not giving guarantee for data consistency.

🡪Its internal data structure is “Doubly Linked List”.

**🡪Wn will be use?**

🡺Best choice for linked list is if our frequent operation is insertion and deletion in the middle.

🡪Linked list is not suitable for retrieval data.

**Methods Used in LinkedList:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1)public void addFirst(object o1);

2)Public void addLast(object o1);

3)Public Object getFirst();

4)Public Object getLast();

5)public object removeFirst();

6)public Object removeLast();

**Difference between ArrayList and LinkedList?**

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses a **resizable array** to store the elements. | 1) LinkedList internally uses a **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory. | 2)Manipulation with LinkedList is **faster** than ArrayList because it used a Doubly linked list, so no bit shifting is required in memory. |
| 3)An ArrayList class can **act as a list** only because it implements List only. | 3)LinkedList class can **act as a list and queue** both because itimplements List and Deque interfaces. |
| 4)ArrayList **is better for storing and accessing** data. | 4)LinkedList is **better for manipulating** data. |

**Vector:**

\*\*\*\*\*\*\*

🡪Jdk 1.0v, legacy collection, direct implementation class to list interface, it is index based, allows duplicate elements, follows insertion order, will not follow sorting order, allows both homogenous and heterogeneous elements, not allows null values, all methods of vector class are synchronized, only one thread allowed at a time, follows sequential order, it will increase execution time, it will reduce application performance, it is giving guarantee for data consistency, it is thread safe.

🡪It’s internal data structure is “Resizable Array”.

🡪it’s initial capacity is 10 elements.

🡪Once the vector reaches maximum capacity the new vector object will be created with new capacity.

New capacity=current capacity \* 2;

🡪It is best choice for frequent retrieval operations, it is not good for frequent insertion and deletion operations.

🡪For deleting and insertion vector is not up to mark because lots of shifting required if we deleting the data in middle.

**Constructors:**

\*\*\*\*\*\*\*\*\*\*\*\*

1)public vector()

2)Public vector(int capacity)

3)public vector(int capacity, int incremental ratio)

4)public vector(Collection c)

**Methods:**

\*\*\*\*\*\*\*\*

1)To add Element

Add(object o1); //collection method

Add(int index, object o1); //List Interface method

Add Element(Object o1); //vector class method

2)To remove Object

Remove Element(object o1);

Remove ElementAt(int index);

Remove AllElements();

3)To retrieve object

Object ElementAt(int index);

Object FirstElement();

Object lastElement();

**Difference b/w ArrayList and Vector?**

|  |  |
| --- | --- |
| ArrayList | Vector |
| Introduced in JDK 1.2v. | Introduced in JDK 1.0v. |
| Is not legacy collection. | Legacy collection. |
| Not synchronized. | Synchronized. |
| No method is synchronized method in ArrayList. | Almost all the methods are synchronized methods in vector. |
| It allows more than one thread at a time to access data. | It allows only one thread at a time to access data. |
| It follows parallel Execution. | It follows sequential Executions. |
| ArrayList is able to reduce the application execution time. | Vector is able to increase application execution time. |
| It is able to improve application performance. | Vector is able to reduce application performance. |
| It is not giving guarantee for data consistency. | It is giving guarantee for data consistency. |
| It is not thread safe. | It is a thread safe. |
| It is incremental capacity is (current capacity\*3/2)+1. | It incremental capacity is 2\*current capacity. |
| We are unable to get capacity value of ArrayList, because no capacity() method in ArrayList class. | We can get capacity value of vector, because, capacity() method is existed in vector class. |

**Stack:**

\*\*\*\*\*\*

Introduced in Jdk 1.0v, it is a legacy collection, it is a chilled class to vector class.

It is a specially designed for Last In First Out(LIFO) (or) It able to arrange as per “Last In First Out”[LIFO] algorithm.

**Methods:**

\*\*\*\*\*\*\*\*

**1)Public Object push(object o):**It will add the specified element to stack (or) to insert the object into the stack.

**2)Public Object pop():**It will remove and return top of the stack.

**3)Public Object peek():**To return the object from the top of the stack.

**4)public Boolean empty():**To return true if stack is empty.

**5)public int search(Object o1):** It will check whether specified element is existed or not in the stack, if the specified element is not existed then it will return ‘-1’ value, if the specified element is existed then it will return it’s position.

|  |  |  |
| --- | --- | --- |
| 0  Off set | A  index | 3 |
| 1 | B | 2 |
| 2 | C | 1 |
| 3 | D | 0 |

**Cursors /Iterators in collections:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1)Enumeration <I>

2)Iterator <I>

3)ListIterator <I>

**1)Enumeration:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

It is a legacy cursor, it is applicable only legacy collections to retrieve elements in one by one fashion.

**(Or)**

We can use enumeration to get an object one by one from the collection. We can create enumeration object by using elements method.

**Method:**

\*\*\*\*\*\*\*

1)Public abstract Enumeration elements();

It can be used only for vector object (or) collection

🡺By using enumeration we can perform only read operations it has two methods.

2)Retrive elements from Enumeration:

a)Check whether more elements are available or not from current cursor position by using the following method

1)public abstract Boolean hasMoreElements()

🡪It will return true value if at least next element is existed.

🡪It will return false value if no element is existed from current cursor position.

b)If at least next element is existed then read next element and move cursor to next position by using

public Object nextElement()

**Drawbacks:**

\*\*\*\*\*\*\*\*\*\*\*

1)Enumeration is applicable for only legacy collections.

2)Enumeration is able to allow only read operation while iterating elements.

Ex:

\*\*\*

Class Test

{

Public static void main(String[] args)

{

ArrayList al=new ArrayList();

Al.add(“AAA”);

Al.add(“BBB”);

Al.add(“CCC”);

Al.add(“DDD”);

Al.add(“EEE”);

Al.add(“FFF”);

System.out.println(al); //[AAA,BBB,CCC,DDD,EEE,FFF]

Iterator itr=al.iterator();

While(itr.hasNext())

{

Object o1=itr.next();

System.out.println(o1);

AAA BBB CCC DDD EEE FFF

}

}

}

o/p:

\*\*\*

AAA

BBB

CCC

DDD

EEE

FFF

**Iterator:**

\*\*\*\*\*\*\*\*

🡪JDK 1.2v, we can use iterator to get an object one by one from the collection.

🡪By using iterator we can able to perform read and remove operations. 🡪We can create an iterator object by using iterator method of collection interface.

Public abstract Iterator iterator

Ex: Iterator itr=al.iterator(); (instead of al we can take anything like hm,tm, …etc).

**Methods in iterator:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1)public abstract Boolean hasNext();

2)public abstract Object next()

3)public abstract void remove()

1)Public abstract Boolean hasNext()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

It will true if the collection has more element. HasNext() method will true object is present where the cursor actively present.

2)public abstract object next()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

It will return the object where the cursor is actively present and it will move the cursor the next index (or) object.

3)public abstract void remove()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

It will remove the object where the cursor is actively present.

Limitations of iterator:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

By using iterator cursor will move forword direction only not backword direction. To overcome this problem we should go for ListIterator.

**Difference between Enumeration and Iterator?**

|  |  |
| --- | --- |
| **Enumeration** | **Iterator** |
| It is a legacy cursor, it was introduced in JDK 1.0v | It is a not legacy cursor, it was introduced in JDK 1.2v |
| It is not universal cursor, it is applicable only legacy collections. | It is an universal cursor, it is applicable for all collection implementations. |
| It is able to allow only read operation while iterating elements. | It is able to allow both read and remove operations while iterating elements. |

**ListIterator:**

\*\*\*\*\*\*\*\*\*\*\*\*

🡪It is an interface to provide jdk1.2v, it able to read elements in both forward and backward directions, it able to allow the operations like read, insert, replace and remove while iterating elements.

🡪By using ListIterator we are perform addition of new object. We can create ListIterator object by using list iterator method.

Public abstract ListIterator listIterator();

ListIterator ltr=al.ListIterator(); (instead of al we can use anything like hm).

**1. Create ListIterator Object:**

public ListIterator listiterator

ex:ListIterator lstr=li.listIterator

**2. Retrive Elements from ListIterator:**

To retrive elements from ListIterator in **Forward Direction** we have to following methods

**public boolean hasNext()**===>It will check next element existed or not from the current cursor

**public Object next()**==>it will next element and it will move cursor to the next position in forward direction

**public int nextIndex(**)==>it will return next index value from the current cursor position

while(lstr.hasNext())

{

s.o.p.l(lstr.nextIndex()+"-->"lstr.next());

}

s.o.p.l();

To retrieve elements in **backward directions**

public boolean hasPrevious();

public Object previous();

public int previousIndex();

while(lit.hasPrevious())

{

s.o.p.l(lit.previousIndex()+"-->"+lit.previous());

To perform operations like remove, insert, replace

public void remove()==>if(element.equals("E")) {lit.remove()}

public void add(Object obj)==>if(element.equals("B")){ lit.add("X") }

public void set(Object obj)==>if(element.equals("C")){ lit.set("Y") }

**Q)diff between Iterator, ListIterator and Enumeration?**

|  |  |  |
| --- | --- | --- |
| **Iterator** | **ListIterator** | **Enumeration** |
| Iterator is applicable for all collection implementations, allowed in forward direction only, is able to allow read and remove operations iterating elements. | ListIterator is applicable for only List implementations, allowed in both forward and back ward directions, is able to allow operations like insert, remove, replace and read operations while iterating elements. | Enumeration is applicable only legacy collection, allowed in forward direction only, allow only read operation while iterating elements. |

**SET:**

\*\*\*\*

jdk1.2v, it is a child interface to collection interface, it is not index based, it able to arrange all the elements on the basis of elements hashcode values, will not allow duplicate elements,

\*\*not follow insertion order(Note: LinkedHashSet follow insertion order),

not follow sorting order(sortedSet, NavigableSet and TreeSet following Sorting order),

\*\*able to allow only one null value(Sorted set, navigable set and Tree set are not allowing even single null value).

When we go for set?

Duplicates not allowed and insertion order will not preserve. Then we should go for set interface.

**HashSet:**

\*\*\*\*\*\*\*\*

HashSet is a direct implementation class to set interface, jdk1.2v, it is not index based, it able to arrange all the elements on the basis of elements hashcode values, will not allow duplicate elements, not follow insertion order, not follow sorting order, able to allow only one null value, internal data structure is "HashTable", initial capacity is 16 elements and initial fill ratio is 75%, not synchronized, all methods are not synchronized in hashset, allows more than one thread at a time, follows parallel execution, will reduce execution time, improves performance of the application, not giving guarantee for data consistency, not thread safe.

🡪Insertion order not preserved and data will a store based on hash code of the object.

🡪Duplicate objects are not allowed. If you trying to insert the duplicate object then won’t give any value. Add method will return false.

**Constructors:**

\*\*\*\*\*\*\*\*\*\*\*\*

1.public Hashset==>can be used to create an empty HashSet object with 16 elements as initial capacity and 75% fill ratio.

HashSet hs=new HashSet();

2.public HashSet(int capacity)==>HashSet hs=new HashSet(20);

3.public HashSet(int capacity, float fill\_ratio)==>

HashSet hs=new HashSet(20,0.85f);

4.public HashSet(collection c)

**LinkedHashSet:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**diff b/w hs and lhs?**

|  |  |
| --- | --- |
| **HashSet** | **LinkedHashSet** |
| Jdk 1.2version | JDK 1.4version |
| Not following insertion order | Following insertion order |
| Internal data structure is “HashTable”. | Internal Data structure is “HashTable” and “LinkedList”. |

EX of HashSet:

\*\*\*\*\*\*\*\*\*\*\*\*

class Test

{

public static void main(String[] args)

{

HashSet hs=new HashSet();

hs.add("B");

hs.add("D");

hs.add("H");

hs.add("A");

hs.add("C");

hs.add("G");

hs.add("E");

hs.add("F");

System.*out*.println(hs);

}

}

O/P:

\*\*\*\*

[A, B, C, D, E, F, G, H]

EX of LinkedHashSet:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class Test

{

public static void main(String[] args)

{

LinkedHashSet lhs=new LinkedHashSet();

lhs.add("B");

lhs.add("D");

lhs.add("H");

lhs.add("A");

lhs.add("C");

lhs.add("G");

lhs.add("E");

lhs.add("F");

System.*out*.println(lhs);

}

}

O/P:

\*\*\*\*\*

B, D, H, A, C, G, E, F

**SortedSet:**

\*\*\*\*\*\*\*\*\*\*

jdk1.2v, it is a child interface to Set interface, not index based, not allowing duplicated Elements, not following insertion order, follows sorting order, allows only homogeneous elements. it is not allowing hetrogeous elements if we are trying to add hetrogeous elements JVM wil rise exception java.lang.ClasscastException. Not allow null values, if we are trying to add null values JVM will rise exception java.lang.NullPointerException. it able to allow only comparable objects, by default we are trying to add java.lang.ClassCastException.

(note: If we are trying to add non comparable objects then we have to use comparator).

**When we are using SortedSet?**

If we want to represent a group of individual objects where duplicates are not allowed and all this objects will be inserted according to some sorting order, then we should go for sorted set.

It will come alphabetic order and numeric means numerical orders will get output.

**Methods:**

\*\*\*\*\*\*\*\*

1.public Object first()==>s.o.p.l(ts.first());

2.public Object last()==>s.o.p.l(ts.last());

3.public SortedSet headSet(Object obj)==>s.o.p.l(ts.headSet("D"));

4.public SortedSet tailSet(Object obj)==>s.o.p.l(ts.tailSet("D"));

5.public SortedSet subSet(Object obj1, Object obj2) ==>s.o.p.l(ts.subSet("B","E"));

**NavigableSet:**

\*\*\*\*\*\*\*\*\*\*\*\*\*

jdk1.6v or java 6 v, it is a child interface to SortedSet interface.

Methods:

\*\*\*\*\*\*\*

1.public Object ceiling(Object obj)==> >=obj1

s.o.p.l(ts.celling("D"));

2.public Object higher(Object obj)==> >obj1

s.o.p.l(ts.higher("D"));==>E

3.public Object floor(Object obj)==> <=obj1

s.o.p.l(ts.floor("D"));==>D

4.public Object lower(Object obj)==> <=obj1

s.o.p.l(ts.lower("D"));==>C

5.public Object pollFirst()==>it will remove 1st element from navigable set==>ts.pollFirst();

6.public Object pollLast()==>it will remove last element from navigable set==>ts.pollLast();

7.public NavigableSet decendingSet()==>S.O.P.L(ts.decendingSet());

**TreeSet:**

\*\*\*\*\*\*\*\*

jdk1.2v, legacy collection, it has provided implementation for collection set, SortedSet and navigableSet. not index based, not allowing duplicated elements, not following insertion order, follows sorting order, allows only homogeneous elements, it is not allowing heterogeneous elements if we are trying to add heterogeneous elements jvm wil rise exception java.lang.ClasscastException.

not allow null values, if we are trying to add null values jvm will rise exception java.lang.NullPointerException.

it able to allow only comparable objects, by default we are trying to add java.lang.ClassCastException.

(note:if wer are trying to add non comparable objects then we have to use java.util.Comparator.

-->it's data structure is "**Balanced Tree**".

-->it is mainly used for frequent search operations).

Constructors:

\*\*\*\*\*\*\*\*\*\*\*\*

1.public TreeSet()==>TreeSet ts=new TreeSet();

2.public TreeSet(Comparator c)

It will create an empty TreeSet object with the explicit sorting mechanisam in the form of comparator.

TreeSet ts=new TreeSet(new MyComparator());

3.public TreeSet(SortedSet ts)-->it will create Treeset object with all elements of the specified SortedSet.

4.public TreeSet(Collection C)

**Queue:**

\*\*\*\*\*\*\*

jdk5.0v, it is a direct child interface to collection interface, it able to arrange all the elements as per FIFO(first come first serve) but it is possible to change this algorithm as per our requirement, able to allow duplicate elements, not following insertion order, following sorting order,

it is not allowing heterogeneous elements if we are trying to add heterogeneous elements JVM will rise exception java.lang.ClasscastException.

🡪Not allow null values, if we are trying to add null values JVM wil rise exception java.lang.NullPointerException.

🡪Able to allow only homogeneous elements.

🡪It able to allow comparable objects by default we are trying to add java.lang.ClassCastException.

(note: if we are trying to add non comparable objects then we have to use comparator. it able to manage all elements prior to process.)

**Wn we use Queue?**

🡪If you want to represent a group of individual objects where the object about the process.

🡪Then we should go for Queue.

🡪LinkedList also implements Queue interface(from 1.5 v).

🡪Queue follows FIFO(First In First Out).

🡪It is used to insert the element at the end of the Queue and remove from the begging of the Queue. Default size is 11.

Methods

\*\*\*\*\*\*\*

1.public void offer(Object obj)-->can be insert to specified element in the queue.(q or pq q.offer("A");)

2.public Object peek()-->can be used to written head element of the queue.(s.o.p.l(q.peek());)

3.public Object element()-->used to written head element of the queue.(s.o.p.l(q.element());)

note:if we access peek() method on an empty queue then peek() will written "null" value.

if we access element() method on an empty queue then element() method will rise an exception like java.util.NoSuchElementException.

4.public Object poll()-->can be used to written and remove head element from queue.(s.o.p.l(q.poll());)

5.public Object remove()-->can be used to written and remove head element from queue.(s.o.p.l(q.remove());)

note: if we access poll() method an empty Queue then poll()method will written "null" value.

if we are access remove() method on an empty queue then remove() method will rise an error like java.util.NoSuchElementException.

**PriorityQueue:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

jdk5.0v, not legacy collection, direct implementation class to queue interface, able to arrange all the elements prior to processing on the basis of priorities, duplicate elements not allowed, not following insertion order, not following sorting order,

not allow null values, if we are trying to add null values jvm wil rise exception java.lang.NullPointerException.

it is not allowing hetrogeous elements if we are trying to add hetrogeous elements jvm wil rise exception java.lang.ClasscastException.

able to allow only homogeneous elements.

🡪It able to allow comparable objects by default we are trying to add java.lang.ClassCastException.

(note: If we are trying to add non comparable objects then we have to use comparator.)

🡪Initial capacity 11 elements, not synchronized, no method is synchronized in PriorityQueue, allows more than one thread at a time to access data.

🡪Follows parallel execution, able to reduce application execution time, able to improve application performance, not giving guarantee for data consistency, not thread safe.

🡪It is used to represent a group of individual objects priority process acc to some priority.

🡪The priority Queue can be either can be either default natural sorting order (or) customized sorting order.

🡪If you are depending on natural sorting order then the object should be homogenous (or) heterogeneous (or) comparable otherwise class cast exception.

🡪If you are depending on own sorting by comparator then object can be homogeneous and heterogeneous.

Constructors:

\*\*\*\*\*\*\*\*\*\*\*\*

1.publicPriorityQueue()-->PriorityQueue pq=new PriorityQueue();

==>s.o.p.l(ts);PriorityQueue pq=new PriorityQueue(ts);s.o.p.l(pq);

2.public PriorityQueue(int capacity)-->

PriorityQueue pq=new PriorityQueue(20);

3.public PriorityQueue(int capacity,comparator c)-->

MyComparator mc=new MyComparator();

PriorityQueue pq=new PriorityQueue(20,mc);

4.public PriorityQueue(sortedSet ss)--> able to create priorityqueue object with all the elements of the specified sortedset.

5.public PriorityQueue(Collection C)

**MAP:**

\*\*\*\*\*

🡪jdk1.2v, it is not child interface to collection interface, it able to arrange all the elements in the form of Key-value pairs. In map, both keys and values are objects. Duplicates are not allowed at keys, but values may be duplicated, only one null value is allowed at keys side, but any number of null values is allowed at values sides.

🡪Both keys and values are able to allow heterogeneous elements, insertion order is not followed, sorting order is not followed.

🡪If you want to represent a group of objects has a key-value pair then we should go for map.

Methods:

\*\*\*\*\*\*\*\*

1.public void put(Object key,object value)-->hm.put("A","AAA"); [A=AAA,B=BBB]--> SINGLE A IS KEY ,AAA IS VALUE

2.public void putAll(Map m)-->hm.putAll(hm1);

3.public Object get(Object key)-->s.o.p.l(hm.get("B"));

4.Public Object remove(Object key)-->s.o.p.l(hm.remove("E"));

5.public int size()-->s.o.p.l(hm.size());-->it will written no of elements in the map

6.public boolean containsKey(Object key)->s.o.p.l(hm.containsKey("B"));

7.public boolean containsValue(Object key)

8.public Set key Set()-->s.o.p.l(hm.keySet());-->A,B,C,D

9.public Collection values()

10.public boolean isEmpty()-->if map is empty it will display true otherwise false.

**Entry:**

**\*\*\*\*\*\***

🡪Each key value pair is called as one entry.

🡪Without existing map object there is no chance of existing entry object.

🡪Hence entry interface is sub class of Map.

Ex:

\*\*\*\*

Interface Map

{

Interface Entry Map.entry<Integer,String>

{

Object getKey();

Object getValue();

Object setValue(object value);

}

}

**HashMap:**

\*\*\*\*\*\*\*\*\*

jdk1.2v, is not legacy, is an implementation class to Map interface, able to arrange all the elements in the form of key-value pairs, both keys and values are objects, duplicates are not allowed at keys side, but any number of null values are allowed at values side, both keys and values are able to allow heterogeneous and homogenous objects, insertion order not followed, sorting order is not followed, internal data structure is "**HashTable**", initial capacity is 16 elements, is not synchronized, no method is synchronized in HashMap, allows more than one thread to access data, follows parallel execution, reduce application execution time, improve application performance, not giving guarantee for data consistency, not thread safe, initial load factory is 75%.

Constructors:

\*\*\*\*\*\*\*\*\*\*\*\*\*

1.public HashMap()

2.public HashMap(int capacity)

3.public HashMap(int capacity,float load factor)

4.public HashMap(Map m)

**LinkedHashMap:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Diff b/w HM and LHM?**

|  |  |
| --- | --- |
| **HashMap** | **LinkedHashMap** |
| JDK 1.2v | JDK 1.4V |
| Not following insertion order | Following insertion order |
| Internal Data Structure is “HashTable”. | Internal Data Structure is “HashTable+LinkedList”. |

**IdentityHashMap:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Diff b/w HM and IHM?**

|  |  |
| --- | --- |
| **HashMap** | **IdentityHashMap** |
| JDK 1.2 | Jdk 1.4v |
| In hashmap JVM will use equals() method to identify the duplicate key which meant for content comparison. | IHM JVM will use double equall operator()[==] to identifier duplicate key which meant for reference comparison. |

**Note:**

\*\*\*\*\*

'==' operator will perform references comparison always, but equals() method was defined in Object class initially, later on it was overriden in String class and in all wrapper classes in order to perform contents comparison.

**WeakHashMap:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Diff b/w HM and WHM?**

|  |  |
| --- | --- |
| **HashMap** | **WeakHashMap** |
| Once if we adding an element to HashMap then HM is not allowing garbage collector to destroy its objects.  (or)  If you are creating HashMap object without reference, still it is not eligible for garbage collector. | Even if we add an element to WHM then WHM is able to allow Garbage collector to destroy elements.  (or)  If you are creating object of weak hash map without reference then it is eligible for garbage collector. |

**Note:**

\*\*\*\*\*

In java applications, garbage collector will destroy objects internally. In java applications, it is possible to destroy objects explicitly by activating GC, for this, we have to use the following steps.

1.nullify the respective object ref.

2.Access System.gc() method, it will access finalize() method internally just before destroying object.

**SortedMap:**

\*\*\*\*\*\*\*\*\*\*

jdk1.2, direct child interface to map interface, able to allow elements in the form of Key value pairs, where both keys and values are objects, will not allow duplicate elements at key side, but it able to allow duplicate elements at values side, will not follow insertion order, will follow sorting order, will not allow null values at key side, if we will try add null values at key side exception like

java.lang.NullPointerException, it will not allow heterogeneous elements at keys side, if we will try JVM will rise exception like

java.lang.ClassCastExceptior, it able to allow only comparable objects at key side by default, if we will try **jvm** will raise exception like

java.lang.ClassCastException. If we want to add non comparable objects then we must use comparator.

Wn we go for sortedMap?

If you want to represent a group of key-value pair and output should be in some sorting order key then we go for SoretedMap.

**Methods:**

\*\*\*\*\*\*\*\*

1.public Object firstKey()-->s.o.p.l(tm.firstKey());

2.public Object lastKey()-->s.o.p.l(tm.lastKey());

3.public SortedMap headMap(Object key)-->s.o.p.l(tm.headMap("D"));

4.public SortedMap tailMap(Object key)-->s.o.p.l(tm.tailMap("D"));

public SortedMap subMap(Object obj1,Object obj2)

-->s.o.p.l(tm.subMap("B","D"));

**NavigableMap:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

It was introduced in java 6, it is a child interface to SortedMap and it has defined methods to provide navigations over the elements.

**Methods:**

\*\*\*\*\*\*\*\*

1.public Object CeilingKey(Object obj)-->s.o.p.l(tm.cellingKey("D"));o/p:D

2.public Object higherKey(Object Obj)-->s.o.p.l(tm.higherKey("C"));o/p:D

3.Public Object floorKey(Object obj)-->s.o.p.l(tm.floorKey("D"));o/p:D

4.public Object lowerKey(Object obj)-->s.o.p.l(tm.lowerKey("C"));o/p:B

5.public NavigableMap descendingMap()

6.public Map.Entry pollFirstEntry()-->s.o.p.l(tm.pollFirstEntry());

7.public Map.Entry pollLastEntry()-->s.o.p.l(tm.pollLastEntry());

**TreeMap:**

\*\*\*\*\*\*\*\*\*

JDK1.2v, not legacy, it is an implementation class to Map, SortedMap and NavigableMap interfaces, it able to allow elements in the form of Key-value pairs, where both keys and values are objects, will not allow duplicate elements at keys side, but it able to allow duplicate elements at values side, not follow insertion order, will follow sorting order, will not allow null values at keys side, if we will try JVM will rise exception like

java.lang.NullPointerException,will not allow heterogenous elements at keys side, if we will try jvm will exception like

java.Lang.ClassCastException,able to only comparable objects at keys side by default, if we will try to add at object side jvm will exception like

java.Lang.ClassCastException.

🡪If we want to add non comparable objects then we must use Comparator.

Its internal data structure is "**Red-Black Tree**(it is child class to balanced key)"-->search in net.

it is not synchronized, no methods are synchronized in TreeMap, it allows more than one thread to access data, it will follow parallel execution,

will reduce execution time, will improve application performance, not giving guarantee for data consistency, it is not thread safe.

**Constructors:**

\*\*\*\*\*\*\*\*\*\*\*\*\*

1.public TreeMap()

2.Public TreeMap(comparator C)

3.Public TreeMap(SortedMap sm)-->we want convert to one map to another map

4.Public TreeMap(Map m)

**Ex of TreeMap:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.pac1;

import java.util.Iterator;

import java.util.Map.Entry;

import java.util.Set;

import java.util.TreeMap;

public class Run

{

public static void main(String[] args)

{

TreeMap<Integer, String> tm=new TreeMap<Integer, String>();

tm.put(102, "Java");

tm.put(104, "HTML");

tm.put(100, "C");

tm.put(105, "CSS");

tm.put(101, "C++");

tm.put(103, "oracle");

tm.put(106, "JavaScript");

System.*out*.println(tm);

tm.put(107, "Angular");

System.*out*.println(tm);

Set<Integer>s1=tm.keySet();

for(Integer key:s1)

{

System.*out*.println(key);

}

System.*out*.println("==using iterator====");

Iterator<Integer> itr=s1.iterator();

while(itr.hasNext())

{

int val=itr.next();

System.*out*.println(val);

}

System.*out*.println("==entry values====");

Set<Entry<Integer, String>> e1=tm.entrySet();

System.*out*.println("==using for each loop===");

for(Entry<Integer, String> en1:e1)

{

System.*out*.println(en1.getKey()+" :"+en1.getValue());

}

System.*out*.println("===using iterator===");

Iterator<Entry<Integer, String>> itr2=e1.iterator();

while(itr2.hasNext())

{

Entry<Integer, String> en2=itr2.next();

System.*out*.println(en2.getKey()+" :"+en2.getValue());

}

}

}

**O/P:**

\*\*\*\*\*

{100=C, 101=C++, 102=Java, 103=oracle, 104=HTML, 105=CSS, 106=JavaScript}

{100=C, 101=C++, 102=Java, 103=oracle, 104=HTML, 105=CSS, 106=JavaScript, 107=Angular}

100

101

102

103

104

105

106

107

==using iterator====

100

101

102

103

104

105

106

107

==entry values====

==using for each loop===

100 :C

101 :C++

102 :Java

103 :oracle

104 :HTML

105 :CSS

106 :JavaScript

107 :Angular

===using iterator===

100 :C

101 :C++

102 :Java

103 :oracle

104 :HTML

105 :CSS

106 :JavaScript

107 :Angular

**Difference b/w HashMap and TreeMap?**

|  |  |
| --- | --- |
| **HashMap** | **TreeMap** |
| HashMap can contain one null key. | TreeMap cannot contain any null key. |
| HashMap maintains no order. | TreeMap maintains ascending order. |

**Hashtable:**

\*\*\*\*\*\*\*\*\*\*

🡪Data structure is “**HashTable**”.

🡪Duplicate keys are not allowed but value can be duplicate.

🡪Insertion order not preserved and it is based on the hash code of key.

🡪Heterogeneous objects are allowed for both key and value.

🡪Null insertions are not possible for both key and value.

🡪Hence will get null pointer exception.

🡪Every method is present in HashTable is synchronized.

🡪Hence HashTable object is thread safe.

**Diff b/w HashMap and Hashtable?**

|  |  |
| --- | --- |
| **HashMap** | **HashTable** |
| Jdk1.2v | Jdk1.0v |
| Not legacy collections | Legacy collections |
| Not synchronized | Synchronized |
| Allows more than one thread to access data | Only one thread at a time to access data |
| Parallel execution | Sequential execution |
| Will reduce execution time | Will increase execution time |
| Will improve application performance | Will reduce application performance |
| Will not give guarantee for data consistency | Will give guarantee for data consistency. |
| Not thread safe | Thread safe |
| In HashMap, one null value is allowed at keys side and any no of null values are allowed at values side. | In case of HashTable, null values are not allowed at both keys and values side. |

Legacy Collection-->java 1.0 means legacy collection, in case jdk 1.2v means it is a Non Legacy Collections.

Synchronized--> it is also 1.0v means synchronized, in case jdk1.2v it is not synchronized.

**Properties<c>:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Properties is a sub class of HashTable. It is used to maintain the least of value in which key is string and the value is string. Multiple threads can share the single properties file.

**Methods of properties file:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1)public String getProperty(String key) 🡪To get the value a specified key.

2)public String setProperty(StringKey,StringValue)

3)public Enumeration propertyName()

4)public void load(InputStream is) 🡺 To load properties file into java properties object.

5)public void store(OutputStream Os) 🡺To store java properties

In java applications, if we any data which we want to change frequently then we have to manage that type of data in a properties file,

otherwise we have to perform recompilation on every modification.

the main purpose of properties files in java application is

1.To manage labels of the GUI components in GUI appl.

2.to manage local irrespective message inI18N appl.

3.to manage exception messages in exception handling.

4.To manage validation messages in data validations.

Ex:

\*\*\*

user.properities

uname=User Name

upwd=User Password

uname.required= User Name is Required

upwd.required=User Password is Required

exception.insufficientFunds= Funds are not sufficient in your account.

In java applications, to represent data of a particular properties file we have to use java.util.Properties class.

**db.properties:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

driver\_Class=sun.jdbc.odbc.jdbc.OdbcDriver

driver\_URL=jdbc:odbc:dsn\_name

db\_user=System

db\_password=saikiran

In collection jdk1.0 and 1.2

|  |  |
| --- | --- |
| JDK 1.0 | JDK 1.2 |
| Legacy | Not Legacy |
| Synchronized | Not Synchronized |
| It allows only one thread at a time to access data. | It allows more than one thread to access data. |
| Sequential Execution | Parallel Execution |
| Will increase Execution time | Will reduce execution time |
| Will reduce application performance | Will improve application performance |
| Will give guarantee for data consistency | Will not give |
| It is Thread safe | It is not Thread safe. |
|  | index based, follows insertion order, not follow sorting order, allows heterogeneous elements.  allows duplicate elements, allow only one null element |
| Ex: Vector, Enumeration, HashTable | EX: Collection, List, ArrayList, LinkedList, Iterator, Set, HashSet, SortedSet, TreeSet, Map, HashMap, SortedMap, TreeMap. |

**Summary of collections:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

java.util package

1.List-->ArrayList,vector,stack,LinkedList

2.Set-->HashSet,Linkedhashset,SortedSet,NavigableSet,TreeSet

3.Queue-->PriorityQueue,BlockingQueue,PriorityBlockingQueue,......

ArrayList-->add, TreeSet-->add, HashSet-->add, vector-->add and addElement, Stack-->push, LinkedList-->add, PriorityQueue-->offer and add,

HashMap-->put, LinkedHashMap-->put, TreeMap-->put.

**Utility classes:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.Collections

2.Arrays

🡪We can sort the elements of set and map.

🡪But we cannot sort the element in list.

🡪Collections classes are provide method for sorting the elements of list type.

🡪All methods present in collections class are static method.

**Methods:**

\*\*\*\*\*\*\*\*

**1)public static void start(list l)**

🡪To sort based on default natural sorting order.

🡪In this case first should contains only homogeneous and comparable object, otherwise we will get runtime exception(class cast exception).

🡪List should not contains null values.

**2)public static void sort(List l, comparator c)**

🡪To sort the list object based on user defined order.

**EX-1:**

\*\*\*\*\*

public class UtilityClass1

{

public static void main(String[] args)

{

ArrayList<Integer> al=new ArrayList<Integer>();

al.add(10);

al.add(25);

al.add(300);

al.add(5);

al.add(510);

al.add(470);

System.*out*.println(al);-------🡪1

Collections.*sort*(al);

System.*out*.println(al);--------🡪2

int index=Collections.*binarySearch*(al, 470);

System.*out*.println(index);------🡪3

}

}

**O/P:**

\*\*\*\*\*

1-🡪[10,25,300,5,510,470]

2-🡪[5,10,25,300,470,510]

3-🡪4

**Searching:**

\*\*\*\*\*\*\*\*\*\*

**1)public static int binarysearch(List l, object target**) 🡺

Default natural sorting order.

**2)public static int binarysearch(List l, object target, comparator c)**

If we are sorting acc to comparator then we should go for this method.

The above method is implemented based on binary search algorithm. Before performing search operation the list should be sorted.

**Ex:**

\*\*\*

Class Test

{

Public static void main(String[] args)

{

ArrayList al=new ArrayList();

Al.add(“z”);

Al.add(“a”);

Al.add(“q”);

Al.add(“k”);

Al.add(“j”);

System.out.println(al); //[z,a,q,k,j]

Collections.sort(al);

System.out.println(al); //[a,j,k,q,z]

Int index=Collections.binarySearch(al,”k”);

System.out.println(index); //2

**2)Arrays <c>:**

\*\*\*\*\*\*\*\*\*\*\*\*\*

**Sorting array element:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Public static void sort(Primitive []p)**

🡪To sort default natural sorting order.

**Public static void sort(Object []o)**

🡪To sort default natural sorting order.

**Public static void sort(Object [], comparator c)**

🡪To sort in customized order.

**Note:**

\*\*\*\*\*

For object type array we can sort acc to natural sorting order (or) customized sorting order.

**Ex:**

\*\*\*

import java.util.Arrays;

public class UtilityClassArray

{

public static void main(String[] args)

{

int[] arr= {15,-40,12,-76,670,95};

for(int i: arr)

{

System.*out*.println(i);-----🡪1

}

Arrays.*sort*(arr);

for(int j :arr)

{

System.*out*.println(j);-----🡪2

}

String []s1= {"sai","kiran","banglore","mechanical","tadipatri","production"};

for(String s:s1)

{

System.*out*.println(s);--------🡪3

}

System.*out*.println("after sorting:");

Arrays.*sort*(s1);

for(String s2:s1)

{

System.*out*.println(s2);-------🡪4

}

}

}

O/P:

\*\*\*\*

**1 2 3 4**

\*\* \*\*\* \*\*\*\* \*\*\*\*\*

15 -76 sai banglore

-40 -40 kiran kiran

12 12 banglore mechanical

-76 15 mechanical production

670 95 tadipatri sai

95 670 production tadipatri

**Arrays:**

\*\*\*\*\*\*\*

**Searching on array element:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1)public static int binarySearch(Primitive [],primitive target)

2)public static int binarySearch(Object []O,Object target)

3)public static int binarySearch(Object[], object target, comparator c)

**Ex-1:**

\*\*\*\*\*

import java.util.Arrays;

public class UtilityClassArraySearch

{

public static void main(String[] args)

{

int[] i= {-230,450,6,-36,52,-310,572};

for(int j:i)

{

System.*out*.println(j);------🡪1

}

Arrays.*sort*(i);

int index =Arrays.*binarySearch*(i, -36);

System.*out*.println(index);------🡪2

char[] ch= {'x','t','l','d','n','c','q'};

for(char ch1:ch)

{

System.*out*.println(ch1);-------🡪3

}

Arrays.*sort*(ch);

For(char ch2:ch)

{

System.*out*.println(ch);-------🡪4

}

int index1=Arrays.*binarySearch*(ch, 'q');

System.*out*.println(index1);--------🡪5

System.*out*.println(Arrays.*binarySearch*(ch, 'l'));------🡪6

}

}

**O/P:**

\*\*\*\*\*

**1 2 3 4 5 6**

\*\*\* \*\*\* \*\*\* \*\*\* \*\* \*\*

-230 2 x c 4 2

450 t d

6 l l

-36 d n

52 n q

310 c t

572 q x

**How to convert array to list?**

Public static list asList(object[] o1)

**Ex:**

\*\*\*

import java.util.Arrays;

import java.util.List;

public class ConvArrToList

{

public static void main(String[] args)

{

String[] s1= {"c","j","q","y","h","u","n"};

List<String> lst=Arrays.*asList*(s1);

System.*out*.println(lst); //[c, j, q, y, h, u, n]

s1[2]="o";

System.*out*.println(l); //[c, j, o, y, h, u, n]

l.set(4, "i");

System.*out*.println(l); //[c, j, o, y, i, u, n]

for(String s2:s1) {

System.*out*.println(s2); --🡪1

}

}

}

1

\*\*\*

c

j

o

y

i

u

n

**Sorting:**

\*\*\*\*\*\*\*

**Comparable<Interface>:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪It is used to order the object of user defined class.

🡪Comparable interface is used in java. lang package

**1)public int compareTo(Object O)**

🡪It is used compare current object with specified object.

**It returns**

\*\*\*\*\*\*\*\*\*

+ve: If the current object is greater than specified object.

-ve: If the object is less than specified object.

0 : If current object and specified object is equal.

**EX:**

\*\*\*

public class Test2

{

public static void main(String[] args)

{

System.*out*.println("A".compareTo("Z")); //-25

System.*out*.println("A".compareTo("z")); //-57

System.*out*.println("a".compareTo("Z")); //7

System.*out*.println("Z".compareTo("K")); //15

System.*out*.println("A".compareTo("A")); //0

}

}

**Note:**

\*\*\*\*\*

If you are using default sorting order like tree set then JVM will call compareTo() method to identify sorting order.

**Ex-1:**

\*\*\*\*\*

import java.util.ArrayList;

import java.util.Collections;

class Employee implements Comparable<Employee>

{

int eid;

String ename;

int esal;

String eaddr;

Employee(int eid, String ename, int esal, String eaddr)

{

this.eid=eid;

this.ename=ename;

this.esal=esal;

this.eaddr=eaddr;

}

@Override

public int compareTo(Employee e1)

{

if(esal<e1.esal)

{

return 1;

}

if(esal>e1.esal)

{

return -1;

}

return 0;

}

}

public class Test

{

public static void main(String[] args)

{

ArrayList<Employee>al=new ArrayList<Employee>();

al.add(new Employee(1247, "sai", 12470, "banglore"));

al.add(new Employee(1650, "kiran", 19650, "chennai"));

al.add(new Employee(5221, "venkat", 72500, "hyderabad"));

al.add(new Employee(5661, "chaithanya", 75620, "banglore"));

al.add(new Employee(6142, "krishna", 65300, "pune"));

Collections.*sort*(al);

for(Employee e1:al)

{

System.*out*.println(e1.eid+","+e1.ename+","+e1.esal+","+e1.eaddr);

}

}

}

**O/P:**

\*\*\*\*

5661, chaithanya, 75620, banglore

5221, venkat, 72500, hyderabad

6142, krishna, 65300, pune

1650, kiran, 19650, chennai

1247, sai, 12470, banglore

**EX-2:**

\*\*\*\*\*

class Employee implements Comparable<Employee>

{

int eid;

String ename;

int esal;

String eaddr;

Employee(int eid,String ename,int esal,String eaddr)

{

this.eid=eid;

this.ename=ename;

this.esal=esal;

this.eaddr=eaddr;

}

@Override

public int compareTo(Employee e1)

{

if(esal<e1.esal)

{

return -1;

}

if(esal>e1.esal)

{

return 1;

}

return 0;

}

}

public class Test

{

public static void main(String[] args)

{

ArrayList<Employee>al=new ArrayList<Employee>();

al.add(new Employee(1247, "sai", 12470, "banglore"));

al.add(new Employee(1650, "kiran", 19650, "chennai"));

al.add(new Employee(5221, "venkat", 72500, "hyderabad"));

al.add(new Employee(5661, "chaithanya", 75620, "banglore"));

al.add(new Employee(6142, "krishna", 65300, "pune"));

Collections.sort(al);

for(Employee e1:al)

{

System.out.println(e1.eid+", "+e1.ename+", "+e1.esal+", "+e1.eaddr);

}

}

}

**O/P:**

\*\*\*\*\*

1247, sai, 12470, banglore

1650, kiran, 19650, chennai

6142, krishna, 65300, pune

5221, venkat, 72500, hyderabad

5661, chaithanya, 75620, banglore

**Comparator<Interface>**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪Comparator interface is used to order the object of user defined class.

🡪It is parent in java.util.package. It contains two methods

**1)public int compare(object o1, object o2)**

🡪It is used to compare current object with specified object(o2).

**2)public Boolean equals(object o1)**

**EX-1:**

\*\*\*\*\*

package com.tekspot.core;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

class Employee

{

String eid;

String ename;

float esal;

String eaddr;

Employe(String eid,String ename,float esal,String eaddr)

{

this.eid=eid;

this.ename=ename;

this.esal=esal;

this.eaddr=eaddr;

}

}

class esalcomparator implements Comparator

{

@Override

public int compare(Object o1, Object o2)

{

Employee e1=(Employee) o1;

Employee e2=(Employee) o2;

if(e1.esal<e2.esal)

{

return 1;

}

if(e1.esal>e2.esal)

{

return -1;

}

else

{

return 0;

}

}

}

class enamecomparator implements Comparator

{

@Override

public int compare(Object o1, Object o2)

{

Employee e1=(Employee) o1;

Employee e2=(Employee) o2;

String ename1=e1.ename;

String ename2=e2.ename;

return ename1.compareTo(ename2);

}

}

public class Run

{

public static void main(String[] args)

{

ArrayList<Employee> al=new ArrayList<Employe1>();

al.add(new Employee("E-111","sai",12000.0f,"Benglore"));

al.add(new Employee("E-222", "venkat", 56000.0f, "Hyderabad"));

al.add(new Employee("E-333","chaitu",75000.0f,"Pune"));

al.add(new Employee("E-444","mahi",15000.0f,"Mumbhai"));

al.add(new Employee("E-555","ajay",50000.0f,"chennai"));

Collections.*sort*(al,new esalcomparator());

for(Employe1 e1:al)

{

System.*out*.println(e1.eid+","+e1.ename+","+e1.esal+","+e1.eaddr);

}

System.*out*.println("compare with ename: ");

Collections.*sort*(al,new enamecomparator());

for(Employe1 e1:al)

{

System.*out*.println(e1.eid+","+e1.ename+","+e1.esal+","+e1.eaddr);

}

}

}

**O/P:**

\*\*\*\*

E-333, chaitu, 75000.0, Pune

E-222, venkat, 56000.0, Hyderabad

E-555, ajay, 50000.0, chennai

E-444, mahi,15000.0, Mumbhai

E-111, sai, 12000.0, Benglore

compare with ename:

E-555, ajay, 50000.0, chennai

E-333, chaitu, 75000.0, Pune

E-444, mahi, 15000.0, Mumbhai

E-111, sai, 12000.0, Benglore

E-222, venkat, 56000.0, Hyderabad

**Ex-2:**

\*\*\*\*\*

package com.tekspot.core;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

class Employee

{

String eid;

String ename;

float esal;

String eaddr;

Employee(String eid,String ename,float esal,String eaddr)

{

this.eid=eid;

this.ename=ename;

this.esal=esal;

this.eaddr=eaddr;

}

}

class esalcomparator implements Comparator

{

@Override

public int compare(Object o1, Object o2)

{

Employee e1=(Employee) o1;

Employee e2=(Employee) o2;

if(e1.esal<e2.esal)

{

return -1;

}

if(e1.esal>e2.esal)

{

return 1;

}

else

{

return 0;

}

}

}

class enamecomparator implements Comparator

{

@Override

public int compare(Object o1, Object o2)

{

Employee e1=(Employee) o1;

Employee e2=(Employee) o2;

String ename1=e1.ename;

String ename2=e2.ename;

return ename1.compareTo(ename2);

}

}

public class Run

{

public static void main(String[] args)

{

ArrayList<Employee> al=new ArrayList<Employe1>();

al.add(new Employee("E-111","sai",12000.0f,"Benglore"));

al.add(new Employee("E-222", "venkat", 56000.0f, "Hyderabad"));

al.add(new Employee("E-333","chaitu",75000.0f,"Pune"));

al.add(new Employee("E-444","mahi",15000.0f,"Mumbhai"));

al.add(new Employee("E-555","ajay",50000.0f,"chennai"));

Collections.*sort*(al,new esalcomparator());

for(Employee e1:al)

{

System.*out*.println(e1.eid+","+e1.ename+","+e1.esal+","+e1.eaddr);

}

System.*out*.println("compare with ename: ");

Collections.*sort*(al,new enamecomparator());

for(Employee e1:al)

{

System.*out*.println(e1.eid+","+e1.ename+","+e1.esal+","+e1.eaddr);

}

}

}

**O/P:**

\*\*\*\*

E-111, sai, 12000.0, Benglore

E-444, mahi, 15000.0, Mumbhai

E-555, ajay, 50000.0,chennai

E-222, venkat, 56000.0,Hyderabad

E-333, chaitu, 75000.0, Pune

compare with ename:

E-555, ajay, 50000.0, chennai

E-333, chaitu, 75000.0,Pune

E-444, mahi, 15000.0, Mumbhai

E-111, sai, 12000.0, Benglore

E-222, venkat, 56000.0, Hyderabad

**EXCEPTIONS**

**\*\*\*\*\*\*\*\*\*\*\***

**Exceptions:**

\*\*\*\*\*\*\*\*\*\*\*

An Exception is unwanted (or) un expected error which occurs during executions of programs i.e at runtime it will disturb the normal flow of program.

**Difference B/W Error and Exception?**

|  |  |
| --- | --- |
| **Error** | **Exception** |
| 🡪An error indicates the problems that any code (or) project should not try to catch. | 🡪An Exceptions indicates the conditions that an application might be catch. |

**Exception Hierarchy:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪All Exception an error type are the sub class of throwable class which is base class of exception hierarchy.

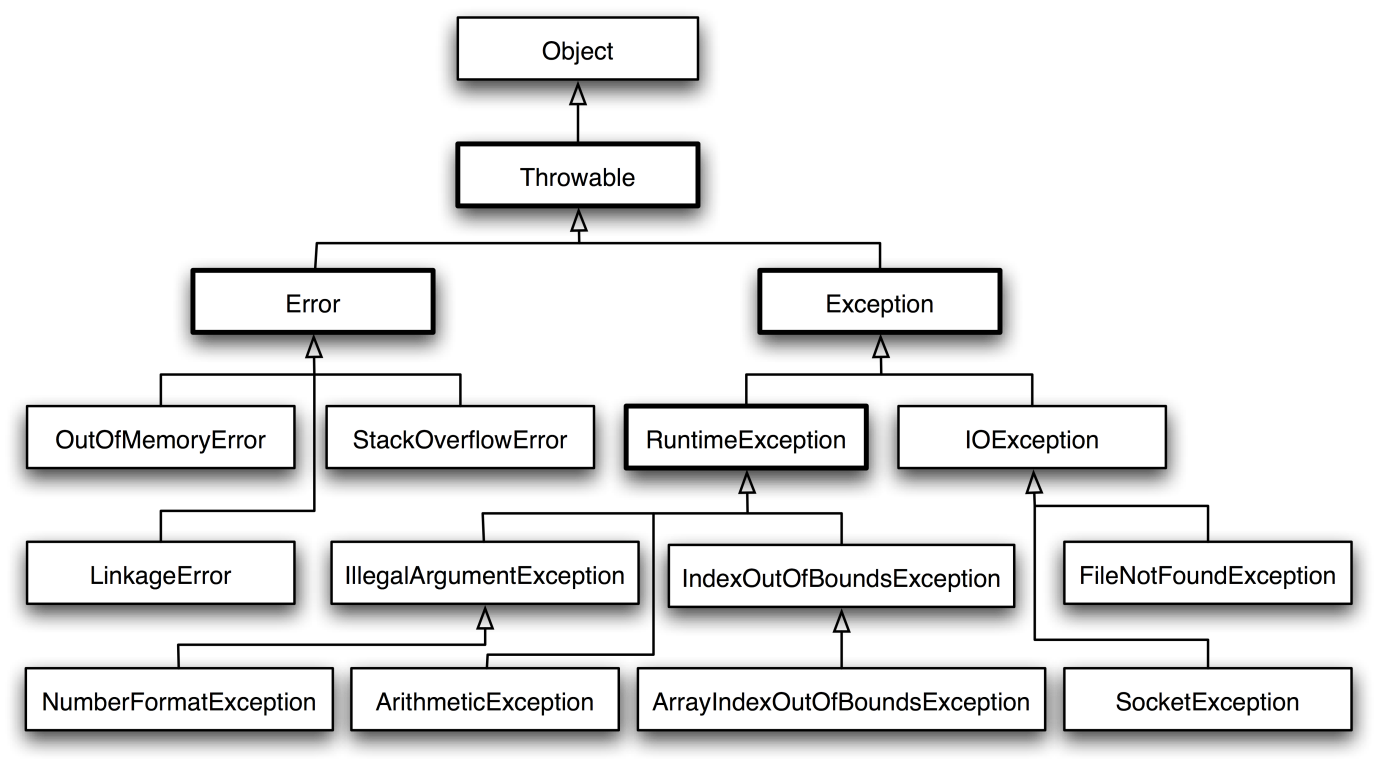
🡪One branch is headed by exception class and this class is for handle the exception that user programs can catch.

**Ex:**

\*\*\*

Arithmetic exception, Nullpointer Exception ………..etc.

🡪Another branch is headed by error class and it is use by JVM at runtime.



There two types of exceptions occurred

1) Checked Exception

2) Un-Checked Exception

**1) Checked Exception:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪It will always occur at the time of compilation and it is checked by compiler.

🡪All checked exceptions it direct class subclass of exception class.

**Ex:**

\*\*\*

IO exception, SQL Exception, class not found exception

**Ex:**

\*\*\*

Class

{

Public static void main(String []args)

Checked Exception

{

FileInputStream fis=new FileInputStream();

}

}

**2) Un-Checked Exception:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪It is not checked at the time of compilation.

🡪It always throw at runtime of execution.

🡪In java under runtime exception class and error class are un-checked exceptions.

🡪Un-Checked exceptions will throw because of programmer mistake.

**Ex:**

\*\*\*

Arithmetic Exception, NullPointer Exception, NumberFormat Exception etc.

**Ex:**

\*\*\*

public class Practice1

{

public static void main(String[] args)

{

int a=10;

int res=a/0;

System.out.println(res); // Arithmetic Exception

String s1=null;

System.*out*.println(s1.length()); // NullPointer Exception

}

}

There are total five ways to handle exception

1) try

2) catch

3) Throw

4) finally

5) try-catch-finally

6) finally-throw

**Flow control in try-catch:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Case-1:**

\*\*\*\*\*\*\*

🡪Exception occur in try block and handle in catch block(un-checked exception).

**Ex-1:**

\*\*\*\*\*

public class Practice1

{

public static void main(String[] args)

{

try

{

int a=10;

int b=0;

System.*out*.println(a/b);

}

catch (ArithmeticException e)

{

e.printStackTrace();

}

System.*out*.println("Rest of the code: ");

}

}

**O/P:**

\*\*\*\*\*

java.lang.ArithmeticException: / by zero

Rest of the code:

**Ex-2:**

\*\*\*\*\*

public class Practice1

{

public static void main(String[] args)

{

try

{

String s1=null;

System.*out*.println(s1.length());

}

catch (NullPointerException e)

{

e.printStackTrace();

}

System.*out*.println("Rest of the code: ");

}

}

**O/P:**

\*\*\*\*\*

java.lang.NullPointerException

Rest of the code:

**Ex-3:**

\*\*\*\*\*

public class Practice1

{

public static void main(String[] args)

{

try

{

String s1=null;

System.*out*.println(s1.length());

}

catch (Exception e)

{

e.printStackTrace();

}

//catch (ArithmeticException e1)

//{

// e.printStackTrace();

//}

//catch (NullPointerException e2)

//{

// e2.printStackTrace();

//}

System.*out*.println("Rest of the code: ");

}

}

**O/P:**

\*\*\*\*\*

java.lang.NullPointerException

Rest of the code:

🡪At a time only one exceptions it can be occurred and at a time only one catch block while executed.

🡪Multiple catch block can be created with single try block.

**Compile Time Error:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪All these catch block exceptions must be ordered from most specific to most generic.

**Ex:**

\*\*\*

public class Practice1

{

public static void main(String[] args)

{

try

{

String s1=null;

System.*out*.println(s1.length());

}

catch (Exception e)

{

e.printStackTrace();

}

catch (ArithmeticException e1) //compile time exception

{

e.printStackTrace();

}

catch (NullPointerException e2)

{

e.printStackTrace();

}

System.*out*.println("Rest of the code: ");

}

}

**Finally block:**

\*\*\*\*\*\*\*\*\*\*\*\*

🡪It is used to execute important code like closing file data base activity.

🡪It will always execute whether the exceptions handle (or) not (or) whether the exceptions occurred (or) not.

🡪We can define nested catch block.

**Ex-1:**

\*\*\*\*\*

public class Practice1

{

public static void main(String[] args)

{

try

{

try

{

String s1=null;

System.*out*.println(s1.length());

}

catch (Exception e)

{

e.printStackTrace();

}

System.*out*.println("Rest of the code 1");

int a=10;

int b=0;

System.*out*.println(a/b);

}

catch (Exception e)

{

e.printStackTrace();

}

System.*out*.println("Rest of the code2: ");

}

}

**O/P:**

\*\*\*\*\*

java.lang.NullPointerException

Rest of the code 1

java.lang.ArithmeticException: / by zero

Rest of the code2:

**EX-2:**

\*\*\*\*\*

public class Practice1

{

public static void main(String[] args)

{

try

{

int a=10;

int b=0;

System.*out*.println(a/b);

}

catch (Exception e)

{

e.printStackTrace();

}

finally

{

System.*out*.println("close file");

}

System.*out*.println("Rest of the code: ");

}

}

**O/P:**

\*\*\*\*\*

java.lang.ArithmeticException: / by zero

close file

Rest of the code:

**Ex-3:**

\*\*\*\*\*

public class Practice1

{

public static void main(String[] args)

{

try

{

int a=10;

int b=10;

System.*out*.println(a/b);

}

catch (Exception e)

{

e.printStackTrace();

}

finally

{

System.*out*.println("close file");

}

System.*out*.println("Rest of the code: ");

}

}

**O/P:**

\*\*\*\*\*

1

close file

Rest of the code:

🡪Finally block it can be executed without catch block.

**Ex-1:**

\*\*\*\*\*

public class Practice1

{

public static void main(String[] args)

{

try

{

int a=10;

int b=0;

System.*out*.println(a/b);

}

finally

{

System.*out*.println("close file");

}

System.*out*.println("Rest of the code: ");

}

}

**O/P:**

\*\*\*\*\*

close file

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

🡪For each try block there can be more than one catch block but only one finally block.

🡪The finally block won’t be executed if the program exit in the middle.

|  |  |  |  |
| --- | --- | --- | --- |
| **In-Valid combinations try-catch block** | | | |
| Try  {  //compiletime error  } | Catch(Exception e)  {  }  Try  {  } | Try  {  }  Finally  {  }  Finally  {  } | Try  {  }  Finally  {  }  Catch (Expection e)  {  } |

**Throw & Throws:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Throws:**

\*\*\*\*\*\*\*

🡪It is a keyword.

🡪It can be used to handle the checked exceptions.

🡪We can declare throws with exception.

🡪It is used to declared with method signature

**Ex:**

\*\*\*

import java.io.FileInputStream;

import java.io.FileNotFoundException;

public class Practice1

{

public static void main(String[] args) throws FileNotFoundException, NullPointerException

{

FileInputStream fis=new FileInputStream("D:\\sai.docx");

}

}

🡪By using throws we can declare more than one exception.

**Throw:**

\*\*\*\*\*\*\*

🡪Java throw keyword is used to create custom exceptions and it will used to explicitly throw an exception.

🡪We can throw either checked (or) un-Checked exceptions.

**Syntax:**

\*\*\*\*\*\*\*

Throw new exception name(String message)

**Ex:**

\*\*\*

Public class Test

{

Public static void main(String[] args) throws FileNotFoundException, NullPointerException

{

checkWebElement(“Jdw”, “Hello”);

}

Public static void checkWebElement(String webelemname, String webelemname)

{

If(webname.equals(name))

{

System.out.println(“Both values are equall: “);

}

Else

{

Throw new NoSuchElementException(“webElement Not found: “ + ”Expected :” +name+” but found:”+webElmname);

}

}

}

**Exception Propagation (or) Default Exception Handler:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪If any method generated an exception and if there is no exception handler then the exception is propagate to below method in the stack call.

🡪If there is an exception handler then it will handle the exception, but if there is no exception handler throughout the program finally there is default exception handler in jre that will handle the default exception.

**Ex:**

\*\*\*

public class Practice1

{

public static void main(String[] args)

{

*m1*();

}

static void m1()

{

System.*out*.println("M1");

}

static void m2()

{

Execution

System.*out*.println("M2");

m4

m3

m2

m1

main

}

static void m3()

Expection

{

System.*out*.println("M3");

}

try

{

*m4*();

}

catch (Exception e)

{

e.printStackTrace();

}

static void m4()

{

System.*out*.println("m4");

System.*out*.println(10/0);

}

}

**O/P:**

\*\*\*\*

M1

M2

M3

M4

**Ex-2:**

\*\*\*\*\*

public class Practice1

{

void m()

{

int data=50/0;

}

void n()

{

m();

}

void p()

{

try

{

n();

}

catch(Exception e)

{

System.*out*.println("exception handled");

}

}

public static void main(String[] args)

{

Practice1 pct=new Practice1();

pct.p();

System.*out*.println("Normal flow......");

}

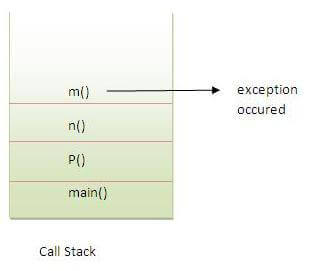
}

**O/P:**

\*\*\*\*\*

exception handled

Normal flow......



🡪By default un-checked exceptions are forwarded into the calling chain.

🡪By default checked exceptions are not forwarded into the calling chain.

**Difference B/W Throw and Throws?**

|  |  |
| --- | --- |
| **Throw** | **Throws** |
| 🡪Java throw keyword is used explicitly throw an exception. | 🡪Java throws keyword used declare an exception. |
| 🡪Throw is used within a method | 🡪Throws is used with method signature. |
| 🡪We cannot create multiple exceptions with single throw keyword. | 🡪But we can declare multiple exceptions with method signature using throws keyword. |

**Difference B/W final, finally and finalize?**

|  |  |  |
| --- | --- | --- |
| **Final** | **Finally** | **Finalize** |
| 🡪Final is a keyword. | 🡪Finally is a block. | 🡪Finalize is a method. |
| 🡪Final is used with class, variable and method and it will use to restrict to change. | 🡪Finally is used to implement with important code, it will always execute whether exceptions are handle (or) not. | 🡪Finalize is a method will be executed just before the object is removed by garbage collect, to disconnect all connections from the object. |
| 🡪Finally block will use with try-catch and try-finally. |

**Abandoned Object:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Object with reference is called as abandoned object.

How to make any object as abandoned object?

Test1 t1=new Test1();

T1=null;

Provide null value to the object reference.

**Exception handling with Method Overriding:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪If super class method does not declare an exception.

a) Super class not declared is an exception then sub class does not declared checked exceptions.

**Ex-1:**

\*\*\*\*\*

class A

{

void m1()

{

System.*out*.println("M1-A");

}

}

class B extends A

{

void m1() throws FileNotFoundException //**compile time error**

{

System.*out*.println("M1-B");

}

}

public class Practice1

{

public static void main(String[] args) throws FileNotFoundException

{

B b=new B();

b.m1();

}

}

**O/P:**

\*\*\*\*\*

Compile time error

b) If the super class method does not declared on exception then sub class overridden method cannot throw checked exception but it can declared unchecked exception.

**Ex-2:**

\*\*\*\*\*

import java.io.FileNotFoundException;

class A

{

void m1()

{

System.*out*.println("M1-A");

}

}

class B extends A

{

void m2() throws FileNotFoundException

{

System.*out*.println("M1-B");

int a=10;

int b=0;

System.*out*.println(a/b);

}

}

public class Practice1

{

public static void main(String[] args) throws FileNotFoundException

{

A a=new B();

a.m1(); ===============🡺 upto here M1-A

B b=new B();

b.m2();

}

}

**O/P:**

\*\*\*\*

M1-A

M1-B

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

🡪If super class method declared an exception.

a) Then subclass overridden method can declared the same exception (or) but sub class does not declared higher exception (or) sub class cannot declared any exception.

**Ex:**

\*\*\*\*

class A

{

void m1() throws ArithmeticException

{

System.*out*.println("M1-A");

int a=10;

int b=0;

System.*out*.println(a/b);

}

}

class B extends A

{

void m1() throws Exception //**compile time error**

{

System.*out*.println("M1-B");

}

}

public class Practice1

{

public static void main(String[] args) throws FileNotFoundException

{

A a=new B();

a.m1();

B b=new B();

b.m1();

}

}

**WRAPPER CLASS**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪The **wrapper class in Java** provides the mechanism to convert primitive into object and object into primitive.

🡪When we create an object to a wrapper class it contains of field and in this field we can a store primitive data types.

🡪We can wrap primitive value into a wrapper class also.

|  |  |
| --- | --- |
| **Primitive data type** | **Wrapper class** |
| Byte | Byte |
| Short | Short |
| Int | Int |
| Long | Long |
| Double | Double |
| Char | Char |
| Float | Float |
| Boolean | Boolean |

🡪The main objective of wrapper classes

i) To wrap the primitive to object form so that we can handle primitive just like an object.

ii) To define several utility methods for primitive like(toString[], valueOf …etc) to perform conversion operations

How to create wrapper class object?

**Constructor:**

\*\*\*\*\*\*\*\*\*\*\*\*

Almost all wrapper class two constructors.

1) For constructor

The primitive values corresponding to wrapper class.

Corresponding string value for wrapper classes.

Integer I1=new Integer(10); 🡪corresponding primitive value of wrapper class

Integer I1=new Integer(“10”); 🡪corresponding String value of wrapper class

Double d1=new Double(10.05);

Double d1=new Double(“10.05”);

🡪If the String argument is not in format of corresponding to the wrapping then we will get in exception like NumberFormatException.

Integer i3=new Integer(“ten”) 🡪Exception

|  |  |
| --- | --- |
| **Wrapper Class** | **Constructor Argument** |
| Byte | Byte, String |
| Short | Short, String |
| Integer | Int, String |
| Long | Long, String |
| Double | Double, String |
| Float | Float, Double, String |
| Character | Char, String |
| Boolean | Boolean, String |

Float f3=new Float(10.05) //accepting double value also

System.out.println(F3);

Boolean b1=new Boolean(true);

System.out.println(b1); //true

Boolean b2=new Boolean(false);

System.out.println(b2); //false

Boolean b3=new Boolean(sai);

System.out.println(b3); //false

Boolean b4=new Boolean(“yes”);

System.out.println(b4); //false

Boolean b5=new Boolean(“No”);

System.out.println(b5); //false

Boolean b6=new Boolean(“true”);

System.out.println(b6); //true

Boolean b7=new Boolean(“false”);

System.out.println(b7); //false

🡪If you are passing Boolean primitive as an argument then the allowed value are true (or) false.

🡪If you are not passing String argument then case is not important and content also not important.

🡪if the content is true (lower case (or) upper case) then it is treated as true value else if false.

Boolean b4=new Boolean (“yes”); //false

Boolean b5=new Boolean(“no); //false

System.out.println(b4.equals(b5)); //true content comparison

**Utility Methods:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1) valueOf()

2) xxxvalue()

3) Parse xxx()

4) toString()

**1) valueOf():**

\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪 We can use value of method to create wrapper class object for the given primitives and string.

🡪It is alternative of constructor.

**Ex:**

\*\*\*

public class Practice1

{

public static void main(String[] args)

{

Integer i1=new Integer(10);

System.*out*.println(i1);

Integer i11=Integer.*valueOf*(10);

System.*out*.println(i11);

Integer i2=Integer.Integer(“10”);---🡪check it

System.out.println(I12);

}

}

**O/P:**

\*\*\*\*\*

10

10

10

**Case-1:**

\*\*\*\*\*\*\*

🡪All wrapper class except character class contains a String value of method to create wrapper class object for the given a string.

**Ex-2:**

\*\*\*\*\*

public class Practice1

{

public static void main(String[] args)

{

Integer i11=Integer.*valueOf*(10);

float f1=Float.*valueOf*("10.05f");

System.*out*.println(i11);

System.*out*.println(f1);

Character ch1=Character.*valueOf*(""); 🡪//error

System.*out*.println(ch1);

}

}

**O/P:**

\*\*\*\*\*

10

10.05

**Form-2:**

\*\*\*\*\*\*\*

Public static wrapper class valueOf(String str, index radix);

🡪Wrapper classes only integer type wrapper class9byte, short, integer, long can have above method).

**Ex:**

\*\*\*

public class Practice1

{

public static void main(String[] args)

{

Integer i11=Integer.*valueOf*("1111",2); -------🡪2o+21+22+23=15

System.*out*.println(i11);

}

}

**O/P:**

\*\*\*\*\*

15

Along radix range is will be 2 to 36

2-🡪 0,1 🡪Binary

8🡪 0 to 9 🡪octal

10 🡪 0 to 9 🡪Decimal

11 🡪 0 to 9, a

12 🡪 0 to 9, a to b

16 🡪 0 to 9, a to F

36 🡪 o to 9, a to z

**XXX value:**

\*\*\*\*\*\*\*\*\*\*

public class Practice1

{

public static void main(String[] args)

{

Integer i=new Integer(30);

System.*out*.println(i.intValue()); //30

System.*out*.println(i.byteValue()); //30

System.*out*.println(i.shortValue()); //30

System.*out*.println(i.longValue()); //30

System.*out*.println(i.floatValue()); //30.0

System.*out*.println(i.doubleValue()); //30.0

}

}

**charValue():**

\*\*\*\*\*\*\*\*\*\*\*\*

To find char primitive for the given character object.

**Ex:**

\*\*\*

public class Practice1

{

public static void main(String[] args)

{

Character ch=new Character('a');

char c1=ch.charValue();

System.*out*.println(c1); //a

}

}

**booleanValue():**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪For the Boolean object if you want to find Boolean primitive then we go for booleanValue().

**Ex-1:**

\*\*\*\*\*

public class Practice1

It will convert character value to character object

{

public static void main(String[] args)

{

Boolean b1=new Boolean("sai");

boolean b=b1.booleanValue();

System.*out*.println(b); **//false**

}

}

**Ex-2:**

\*\*\*\*\*

public class Practice1

{

public static void main(String[] args)

{

Boolean b1=new Boolean(true);

boolean b=b1.booleanValue();

System.*out*.println(b); **//true**

}

}

XXX value()

primitive

Wrapper object

**Parsexxx:**

\*\*\*\*\*\*\*\*\*

**Case-1:**

\*\*\*\*\*\*\*

**Public static primitive parseXXX(String str):**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪The given a string to find primitive values.

🡪For every wrapper class expect character class contains a static parse xxx method to find primitive value for the given object.

**Ex:**

\*\*\*\*

public class Practice1

{

public static void main(String[] args)

{

int i=Integer.*parseInt*("20");

System.*out*.println(i); //20

double d=Double.*parseDouble*("25.36");

System.*out*.println(d); //25.36

boolean b=Boolean.*parseBoolean*("saikiran");

System.*out*.println(b); //false

}

}

**Case-2:**

\*\*\*\*\*\*\*

Public static primitive parseXXX(String str, int radix);

🡪This method will be available only in integer type wrapper classes (byte, short, int, long).

**Ex:**

\*\*\*\*

public class Practice1

{

public static void main(String[] args)

{

int i=Integer.*parseInt*("1111", 2);

System.*out*.println(i); //**15**

int i1=Integer.*parseInt*("10", 2);

System.*out*.println(i1); //**2**

}

}

Parse XXX

primitive

String

**toString():**

\*\*\*\*\*\*\*\*\*\*

🡪We can use toString method to convert wrapper object (or) primitive to String.

**1) public string toString():**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

🡪it is a non-static method.

**Ex:**

\*\*\*\*

public class Practice1

{

public static void main(String[] args)

{

Integer i=new Integer(25);

String s1=i.toString();

System.*out*.println(s1); //25

}

}

**2) public static string toString(primitive p):**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public class Practice1

{

public static void main(String[] args)

{

String s1=Integer.*toString*(25);

System.*out*.println(s1); //25

String s2=Double.*toString*(31.26);

System.*out*.println(s2); //31.26

String s3=Character.*toString*('S');

System.*out*.println(s3); //S

}

}

**3)public static String toString(primitive p, int radix):**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**EX:**

\*\*\*\*

public class Practice1

{

public static void main(String[] args)

{

String s1=Integer.*toString*(15, 2);

System.*out*.println(s1);

String s2=Long.*toString*(22341500, 4);

System.*out*.println(s2);

}

}

**O/P:**

\*\*\*\*\*

1111

1111032131330

**Boxing and Un-Boxing:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**AutoBoxing:**

\*\*\*\*\*\*\*\*\*\*\*

🡪Automatic conversion of primitive type to object of their corresponding wrapper classes is known as autoboxing.

**Ex:**

\*\*\*\*

Int to Integer

Char to Character

Long to Long

**Ex-1:**

\*\*\*\*\*

public class Practice1

{

public static void main(String[] args)

{

char ch='S';

Character ch1=ch;

System.*out*.println(ch1); //S

int i=51;

System.*out*.println(i); //51

Integer it=i;

System.*out*.println(it); //51

}

}

**Ex-2:**

\*\*\*\*\*

public class Practice1

{

public static void main(String[] args)

{

ArrayList al=new ArrayList();

al.add(10);

al.add(20);

here converting primitive 10, 20, 30, 40, 50 to Integer value

al.add(30);

al.add(40);

al.add(50);

System.*out*.println(al);

}

}

**O/P:**

\*\*\*\*\*

[10, 20, 30, 40, 50]

**Un-Boxing:**

\*\*\*\*\*\*\*\*\*\*

🡪Converting the object of wrapper class to it’s corresponding primitive type.

Integer to int

Long to long

Character to char

**Ex-1:**

\*\*\*\*\*

public class Practice1

{

public static void main(String[] args)

{

ArrayList<Integer> al=new ArrayList<Integer>();

al.add(10);

al.add(20);

al.add(30);

al.add(40);

al.add(50);

Autocasting from int to Integer

System.*out*.println(al);

int i=al.get(2);

Unboxing from integer to int

System.*out*.println(i);

}

}

**O/P:**

\*\*\*\*\*

[10, 20, 30, 40, 50]

30

Constructor

Wrapper class

Primitive

valueOf()

XXX value()

Wrapper class

Primitive

Parse xxx()

Primitive

String

toString()

String

Primitive