Java Programming

Basic -   
Current – 20+

Most Popular – Java 8 and Java 16 ( Spring 2+), Java 17 ( Spring boot 3+)

JDK vs JRE -> Java Development Kit and Java Run time Environment

Features :

Simple, Platform Independent, Robust Secure, Multi threading

IDE – Intellij IDEA Community Version, Eclipse , STS

Setting environment variables

We ran sample program ( HelloWorld) -> Create a sample program and run

public class HelloWorld {  
  
 String name;  
   
 String firstName;  
 String lastName;  
   
 Integer age;  
 String address;  
  
 static String *city*;  
 static String *state*;  
  
 static Integer *zipCode*;  
 static {  
 *city* = "Illinio";  
 *state* = "Ohio";  
 *zipCode* = 10;  
 }  
  
 //Naming Variable  
   
  
 public static void main(String[] args) {  
 System.*out*.println("Hello World, Welcome Java programming!!");  
 //Creating an object  
 HelloWorld obj1 = new HelloWorld();  
 obj1.name = "Rahul";  
 obj1.age = 12;  
 obj1.address = "sample address";  
 System.*out*.println("Name : " +obj1.name);  
 System.*out*.println("Age : " +obj1.age);  
 System.*out*.println("Address : " +obj1.address);  
  
 //Invoking a method using object  
 obj1.display();  
 obj1.display2();  
  
 //invoking a static method  
 HelloWorld.*display3*();  
 System.*out*.println("City : " +HelloWorld.*city* + ", State " +HelloWorld.*state*+ ", Zip code : "+HelloWorld.*zipCode*);  
 }  
  
 public static void main(String[] args, Integer a) {  
 System.*out*.println("Hello World, Welcome Java programming!!");  
 }  
  
 //static methods are associated with classes  
 public static void main(String[] args, boolean flag) {  
 System.*out*.println("Hello World, Welcome Java programming!!");  
 }  
  
 //non-static method are associated with objects  
 public void display() {  
 System.*out*.println("I am inside display method");  
 }  
  
 //non-static method are associated with objects  
 public void display2() {  
 System.*out*.println("I am inside display method 2");  
 }  
  
 public static void display3() {  
 System.*out*.println("I am inside display method 3 and it is static method");  
 }  
  
   
  
}

Access Modifiers and Local Variables  
package training.basics;  
  
public class AccessModifier {  
  
  
 private int data = 50;  
  
 public static void main(String[] args) {  
  
 //Java Variable - Local Variables, Class Variables( Static variables) , Instance variables ( non static variables)  
 String localVariable1 = "I am local variable";  
 System.*out*.println("Local variable : " +localVariable1);  
  
 AccessModifier accessModifier = new AccessModifier();  
 accessModifier.displayMethod1();  
 System.*out*.println("Accessing private variable : " +accessModifier.data);  
  
 //Access Modifier - public, private , protected and default  
  
 }  
  
 public void displayMethod1() {  
  
 //Local variable  
 Integer var1 = 100;  
 System.*out*.println("inside display method 1 , var1 : " +var1);  
 }  
  
}

Constructors

public class Constructor {  
 String name;  
 Integer age;  
  
 public Constructor() {  
 System.*out*.println("I am inside constructor");  
 this.name = "Rahul";  
 this.age = 10;  
 }  
  
 public Constructor(String name, Integer age) {  
 System.*out*.println("I am inside parameterized constructor");  
 this.name = name;  
 this.age = age;  
 }  
  
 public static void main(String[] args) {  
 System.*out*.println("i am inside main");  
 Constructor constructor = new Constructor();  
 System.*out*.println("Name : " + constructor.name + ", age : " + constructor.age);  
 Constructor constructor1 = new Constructor("Mohan", 12);  
 System.*out*.println("Parameterized constructor values : Name - " + constructor1.name + ", Age : " + constructor1.age);  
 }  
}

Inheritance

package training.inheritance;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
  
public class Department {  
  
 String departmentName;  
 String departmentCode;  
  
 public List<String> getEmployee() {  
  
 System.*out*.println("i am inside getEmployee method of Department class");  
 List<String> employees = new ArrayList<>();  
 employees.add("emp1");  
 employees.add("emp2");  
 employees.add("emp3");  
 employees.add("emp4");  
 return employees;  
 }  
  
  
}

package training.inheritance;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class Finance extends Department {  
  
 public static void main(String[] args) {  
 Department department = new Department();  
 department.departmentCode = "DEPARMENT\_CODE";  
 department.departmentName = "DEPARTMENT";  
  
 List<String> departments = department.getEmployee();  
  
 System.*out*.println("Department list : " + departments);  
  
 Finance finance = new Finance();  
 List<String> financeEmployees = finance.getEmployee();  
 System.*out*.println("Finance list : " + financeEmployees);  
  
 Department department1 = new Finance();  
 department1.departmentName = "FINANCE";  
 department1.departmentCode = "FINANCE\_CODE";  
 System.*out*.println("Department Name " +department1.departmentName);  
 System.*out*.println("Department Code " +department1.departmentName);  
 System.*out*.println("Calling getEmployee method - " +department1.getEmployee());  
 }  
  
 @Override  
 public List<String> getEmployee() {  
 System.*out*.println("i am inside getEmployee method of Finance class");  
 List<String> employees = new ArrayList<>();  
 employees.add("emp3");  
 employees.add("emp4");  
 employees.add("emp5");  
 employees.add("emp6");  
 return employees;  
  
 }  
  
}

**Polymorphism**

1. Method Overloading ( Static Polymorphism/Compile Polymorphism )
2. Method Overridding ( Dynamic Polymorphism/Runtime Polymorphism )

**Method Overloading (Static Polymorphism/Compile Polymorphism )**

package training.polymorphism;  
  
public class MethodOverloadingExample {  
  
 public Integer sum(int a, int b) {  
 System.*out*.println("i am inside sum method with 2 arguments");  
 Integer result = a + b;  
 return result;  
 }  
  
 public Integer sum(int a, int b, int c) {  
 System.*out*.println("i am inside sum method with 3 arguments");  
 Integer result = a + b + c;  
 return result;  
 }  
  
 public Integer sum(int a, int b, int c, int d) {  
 System.*out*.println("i am inside sum method with 4 arguments");  
 Integer result = a + b + c + d;  
 return result;  
 }  
  
 public Double sum(double a, double b) {  
 System.*out*.println("i am inside sum method with 2 arguments having double value");  
 Double result = a + b;  
 return result;  
 }  
  
 public static void main(String[] args) {  
 MethodOverloadingExample obj = new MethodOverloadingExample();  
  
 //Other way of calling a method and printing the result  
 System.*out*.println("Sum of 2 numbers : " +obj.sum(3,5));  
  
 System.*out*.println("Sum of 3 numbers : " +obj.sum(3,5,6));  
  
 System.*out*.println("Sum of 4 numbers : " +obj.sum(3,5,6, 6));  
  
 System.*out*.println("Sum of 4 numbers : " +obj.sum(2.5,3.5));  
 }  
  
  
  
  
  
}

**Method Overridding (Dynamic Polymorphism/Runtime Polymorphism )**

**Abstraction : Two way to achieve abstraction in Java**

1. **Abstract class**
2. **Interface**

**Abstract class:**

It can have abstract and non-abstract methods.

It can have constructor and static methods.

Abstract classes are partially abstract.

**Interface:**

It can have static and final variables.

Interfaces are fully abstract.

It can have only abstract methods until Java 7 Version

Post java 8, interface can also have default and static methods as well.

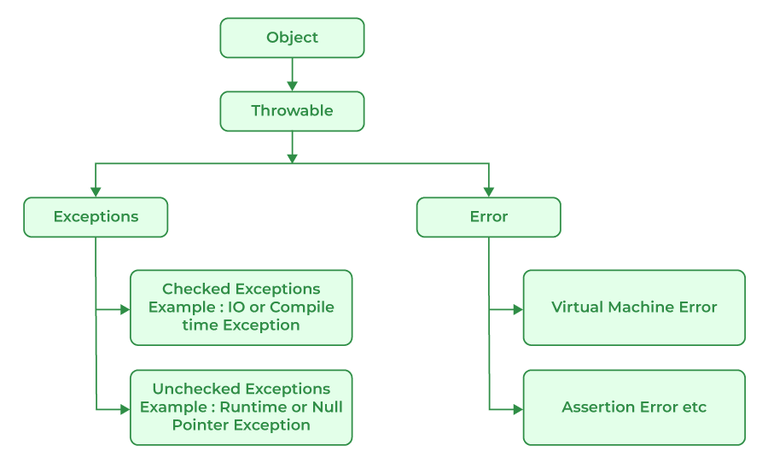
Wrapper Classes

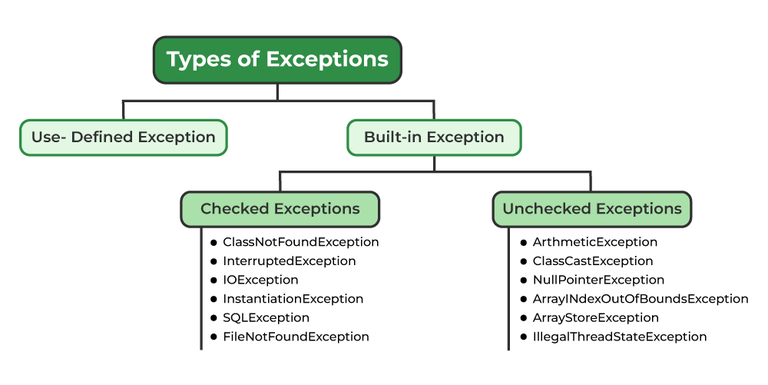
A table with different types of wrapper class

Description automatically generated

| **StringBuffer Class** | **StringBuilder Class** |
| --- | --- |
| StringBuffer is present in Java. | StringBuilder was introduced in Java 5. |
| StringBuffer is synchronized. This means that multiple threads cannot call the methods of StringBuffer simultaneously. | StringBuilder is asynchronized. This means that multiple threads can call the methods of StringBuilder simultaneously. |
| Due to synchronization, StringBuffer is called a thread safe class. | Due to its asynchronous nature, StringBuilder is not a thread safe class. |
| Due to synchronization, StringBuffer is lot slower than StringBuilder. | Since there is no preliminary check for multiple threads, StringBuilder is a lot faster than StringBuffer. |

Exceptions in Java





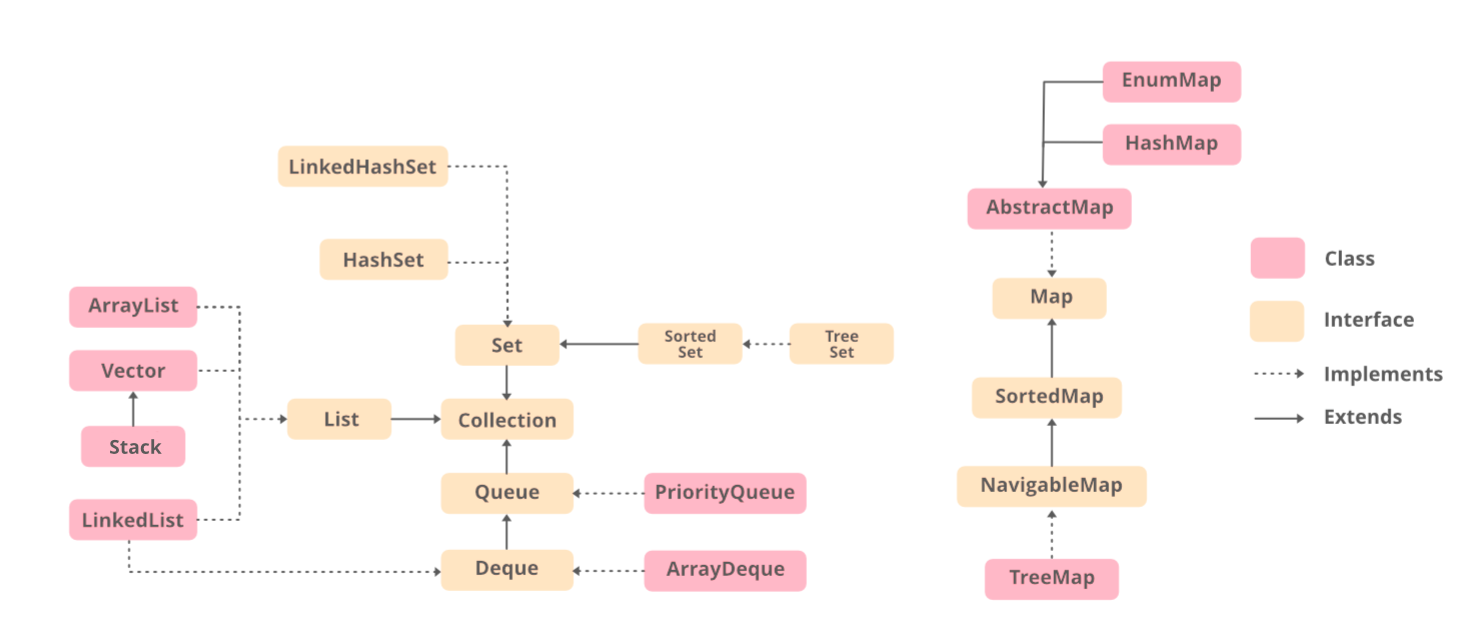
Built in Exceptions:

1. Checked Exception
2. Unchecked Exception

Different ways to throw an Exception

1. Using try catch block
2. Using throws key word which we need to use at method level

Collections in Java



Extends

Class Name

Class Name

Interface

Interface

Class Name

Implements

Interface

extends

**List :** Stores elements(objects) in an ordered format and allows duplicates elements

ArrayList and LinkedList

1. Add()
2. Remove()
3. Contains()
4. Iterating over list

**Set:** Store elements in an un ordered format and doesn’t allow you to store duplicate elements

1. **HashSet**
2. **TreeSet**
3. **CopyWriteArraySet**
4. **LinkedHashSet**
5. **Differences Between HashSet, LinkedHashSet**,**and TreeSet:**

| **Features** | **HashSet** | **LinkedHashSet** | **TreeSet** |
| --- | --- | --- | --- |
| Internal Working | HashSet internally uses HashMap for storing objects | LinkedHashSet uses LinkedHashMap internally to store objects | TreeSet uses TreeMap internally to store objects |
| When To Use | If you don’t want to maintain insertion order but want to store unique objects | If you want to maintain the insertion order of elements then you can use LinkedHashSet | If you want to sort the elements according to some Comparator then use TreeSet |
| Order | HashSet does not maintain insertion order | LinkedHashSet maintains the insertion order of objects | While TreeSet orders the elements according to supplied Comparator. By default, objects will be placed according to their natural ascending order. |
| Complexity of Operations | HashSet gives O(1) complexity for insertion, removing, and retrieving objects | LinkedHashSet gives insertion, removing, and retrieving operations performance in order O(1). | While TreeSet gives the performance of order O(log(n)) for insertion, removing, and retrieving operations. |
| Performance | The performance of HashSet is better when compared to LinkedHashSet and TreeSet. | The performance of LinkedHashSet is slower than TreeSet. It is almost similar to HashSet but slower because LinkedHashSet internally maintains LinkedList to maintain the insertion order of elements | TreeSet performance is better than LinkedHashSet except for insertion and removal operations because it has to sort the elements after each insertion and removal operation. |
| Compare | HashSet uses equals() and hashCode() methods to compare the objects | LinkedHashSet uses equals() and hashCode() methods to compare it’s objects | TreeSet uses compare() and compareTo() methods to compare the objects |
| Null Elements | HashSet allows only one null value. | LinkedHashSet allows only one null value. | TreeSet does not permit null value. If you insert null value into TreeSet, it will throw NullPointerException. |
| Syntax | HashSet obj = new HashSet(); | LinkedHashSet obj = new LinkedHashSet(); | TreeSet obj = new TreeSet(); |

**Decision Making**

1. **If**

**Syntax:**

**if(condition) {**

**}**

1. **If -else**

**Syntax : if(condition) {  
 //statement**

**} else {  
 //statement  
}**

1. **Nested-if  
   if(condition) {  
    if( condition) {  
    //statement  
    }  
   } else {  
    if(condition) {  
    //statement  
    }  
   }**
2. **If-else-if  
   Syntax : if(condition) {  
    //statement**

**} else if(condition) {  
//statement  
} else {  
 //statement  
}**

1. **Switch case :**

**Loops in Java:**

1. While loop
2. Do while loop
3. For loop

While Loop: syntax

/\*   
while ( condition ) {  
 // statements  
}\*/

For Loop : syntax

for (initialization condition; testing condition;increment/decrement)

{

statement(s)

}

**Syntax:**

do

{

statements..

}

while (condition);

Java 8 :

1. Stream API
2. Functional Interface & Lambda Functions
3. Method Reference
4. Optional Class

Stream API: There are two operations.

1. Intermediate operation -> filter(), map()
2. Terminal Operation -> forEach(), distinct(), collect(), max(), min()

Lambda Expression:

Zero Parameters : () -> System.out.println(“No Parameters”);

One Parameter: s -> System.out.println(“One Parameters ” +s);

Multiple Parameters: (int x, int y) -> x +y;

Functional Interfaces:

* An interface that contains only one abstract method ( you don’t need mention method as abstract)
* You can **default and static methods**, but can contain only one abstract method
* Functional Interface is also known as Single Abstract Method Interface
* Its one of the new feature of Java 8
* Functional Interface can extend another interface only when it does not have any abstract method

In Built functional interface in Java 8

* Predicate
* Consumer
* Supplier
* Function
* Comparator
* Comparable
* Runnable

Optional Class : This was class introduced in order to handle NullPointerException.