**Java Basics**

* Define the scope of variables.
* Define the structure of a Java class
* Create executable Java applications with a main method; run a Java program from the command line; produce console output
* Import other Java packages to make them accessible in your code
* Compare and contrast the features and components of Java such as: platform independence, object orientation, encapsulation, etc.

.java (platform independent) 🡪 compile (platform independent) 🡪 bytecode (platform independent) 🡪 jvm 🡪 machine code (platform dependent)

JDK – JRE + development tool

JRE – JVM + library classes

JVM –

Int a = 10 ; a – variable, 10 – literal

**A screenshot of a computer

Description automatically generated**

By default Integral value is int

Decimal - 0 to 9 , int x = 10;

Octal - 0 to 7, must start with 0, int x = 010

Hexadecimal - 0 to 9, a to f - 0X10; Note: case sensitivity does not matter in hexadecimal value

Note: there is no way to assign explicitly a byte type or short type.

Byte b = 100b; - invalid , Short s = 100s; - invalid

To Decimal we can assign each type of value – octal, hexadecimal, decimal.

By default, floating value is double

Note: floating point literal only accept decimal , hexadecimal and octal can’t be assigned to the floating point literals.

In other case, integral literal, octal decimal and hexa can be assigned to integral literals.

Exponential form – it is always double

Double d = 1.2e3;  1.2\*10^3  1.2\*1000 1200.0

Byte 🡪 short 🡪

Int🡪 long🡪 float🡪 double

Char🡪

Int [4] x; - not allowed at the time of declaration we are not allowed to specify the size.

Note: if you want to specify dimension before variable this facility is only available for the first variable

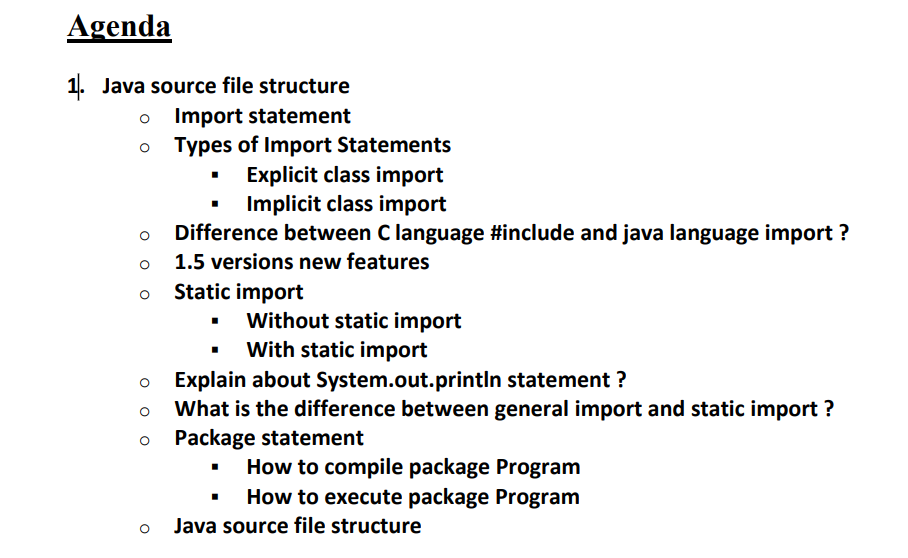
Int [] x = new int [3];

Sop(x); // [[1@hashcode

Int []x;

X = {10,20,30} // invalid – all the decleration should be in single line for shortcut method

8. Declaration and Access Modifiers



**Done all**

Note: - 1. Program contain any number of class

2. If none of the class is public then source file can be anything

3. But, if class contain public class then source file name must be same as public

4. one source file can contain atmost one public class

5. If both B and C classes are declared as public and name of the file is B.java then we will get compile time error saying "class C is public, should be declared in a file named C.java".

6. It is highly recommended to take only one class for source file and name of the Program (file) must be same as class name. This approach improves readability and understandability of the code.

7. for each java class one .class file will be created on compile using javac <filename>

8. if the class has main method then it will run otherwise " "main" java.lang.NoSuchMethodError: main

**Import statement:**

* 1. Explicit
  2. Implicit

If two import packages have same class then it give **error: reference is ambiguous**

**D:\Java>javac Test.java Test.java:7: reference to Date is ambiguous; both class java.sql.Date in java.sql and class java.util.Date in java.util match**

While resolving class names compiler will always gives the importance in the following order.

1. Explicit class import

2. Classes present in current working directory.

3. Implicit class import.

**Whenever we are importing a package all classes and interfaces present in that package are by default available but not sub package classes.**

In any java Program the following 2 packages are not require to import because these are available by default to every Java Program.

1. java.lang package

2. default package (current working directory)

"Import statement is totally compile time concept" if more no of imports are there then more will be the compile time but there is "no change in execution time".

import static java.lang.Math.sqrt; if we have so many static variable to use .. so using class name every time is not feasible

While resolving static members compiler will give the precedence in the following order.

1. Current class static members

2. Explicit static import

3. Implicit static import.

**NOTE:** for normal import we import till class - **import java.lang.Math;**

But for static import we import till method and variable - **import static java.lang.Math.sqrt;**

package com.durga;

class Test

{

public static void main(String[] args)

{

System.out.print("first class");

}

}

To run it via command line if package is available

* Javac Test.java
* Java com.durga.Test **// java Test (if package is not available)**

2 package statement is not allowed.

In any java Program the 1st non comment statement should be package statement [if it is available] otherwise we will get compile time error.

**Order of programme:**

Package – single only

Import- any number of

Class/interface/enum/decleration – any number of

**\*\*\* what valid java program can contain**

* + - Only package - valid
    - Only import - valid
    - Only package and import - valid
    - Only class - valid
    - Both package , import and class - valid

**CHAPTER 5 – OOPS:**

**Data Hiding:**  - Data hiding in Java is restricting access to certain components using access modifiers.

**Encapsulation:** bundling data and methods that operate on that data into a single unit, called a class.

**Difference between Data Hiding and Encapsulation**

Data hiding refers to the practice of restricting access to certain data members or methods of a class so that they can only be accessed and modified by the class itself and not by external code. On the other side, Encapsulation is a broader concept that refers to the practice of bundling data and methods that operate on that data into a single unit, called a class.

**Abstraction**: Hiding the internal implementation and Highlights the set of services which is offering.

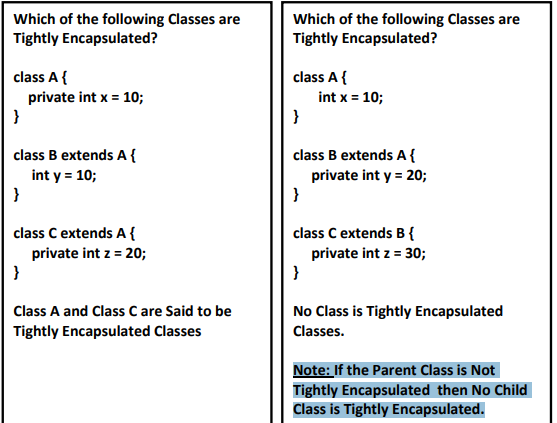
**Encapsulation = Data Hiding + Abstraction**

The Main Advantage of Encapsulation is we can Achieve Security and the Main Disadvantage of Encapsulation is it

**Tightly Encapsulated Class:**

Tightly Encapsulated Class:

A Class is Said to be Tightly Encapsulated if and only if Each and Every Variable of that Class declared as private.



**IMP:**

Interface extends other interface

Class extends other class

Class implements other interface

Interface can’t extend or implement class

An interface cannot extend a class but it can extend another interface in the same way that a class can extend another class.

Type of Inheritance allowed:

1. Single inheritance -allowed
2. Multi-level-inheritance - allowed
3. Multiple inheritance – not allowed ()

A black and white image of a rectangular object with text

Description automatically generated with medium confidence

1. Cyclic inheritance – not allowed

A close-up of a diagram

Description automatically generated

IS-A – one class extends another class

Has-A relationship – new object using new keyword is used

It is also know as composition or Aggregation.

**Is-A relationship: Whenever one class inherits another class, it is called an IS-A relationship. Has-A relationship: Whenever an instance of one class is used in another class, it is called HAS-A relationship.**

**Composition** – existence of contained object is depend on container object and they are strongly depend on each other

**Aggregation** - existence of container object is not depend on container object, loose coupling.

**Note:**

In Composition Objects are Strongly Associated whereas in Aggregation Objects are Weakly Associated. • In Composition Container Object holds Contained Objects Directly whereas in Aggregation Container Object Just Holds References of Contained Objects.

Example: composition

Class engine

{

}

Class car

{

Engine e;

Car()

{

this.e = new Engine(); - Tightly coupled;

}

}

Example: Aggregatopm

Class engine

{

}

Class car

{

Engine e = new Engine(); - loosely coupled

Car()

{

}

}

Method Signature: - Name (argument type)

Return type is not a part of method signature

Compiler will Use Method Signature while resolving Method Calls.

Overloading: same method name and

different argument type or

different order of argument

Compiler take decision about the overloading as it is based on method signature

So it is compile time polymorphism:

Case 1: - The overloading method resolution is responsibility of compiler based on Reference Type and method Argument. It is called Compile Time Polymorphism.

Case 2: Automatic promotion in overloading

Case 3: In overloading exact match will get High Priority.

* In overloading child class argument will get more priority than parent class argument.

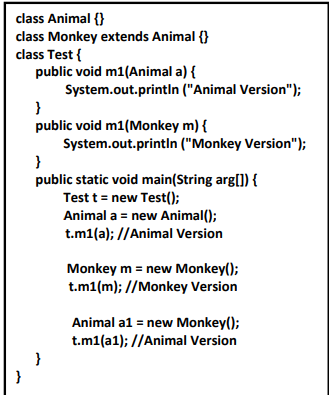
Case 4: in Java, method overloading is not possible by changing the return type of the the method because they may occur ambiguity problem.

Case 5: var-args and array for same type can’t be write together.

* Array get more priority then var – args

Case 6: change in order of argument is also acceptable for overloading

Case 7:Overloading method resolution will always take care by compiler based on Reference Type but not based on Runtime Object.



Overriding: - runtime polymorphism (based on run time object)

Parent class method is by default available to child class

If the child class is not satisfied with the parent class implementation, then the child is allowed to redefine that method in the child class based on its requirement.

In overriding **Method Resolution Always Take care by JVM based on Runtime Object.**

Hence Overriding is considered as Runtime Polymorphism OR Dynamic Polymorphism or Late Binding.

The Parent Class Method which is Overridden is called Overridden Method and the Child Class Method which is Overriding is called Overriding Method.

Case 1:

In Overriding, Method Names and Argument Types Must be Same. i.e Method Signatures Must be Same

Case 2: In Overriding the Return Types Must be Matched

Or

Co- Variant Return Types are allowed.

**Note: Co-Varient Return Type Concept Applicable Only for Object Types but Not for Primitives.**

Covariant return types are implemented in Java by allowing a subclass method to override a superclass method and return a subtype of the superclass method's return type. This means that the return type of the overriding method must be a subtype of the return type of the overridden method.

Case 3: Private method is not visible in the child classes

**Note:** but based on requirement we can define exactly same private method in child class. It is valid but not overriding.

Case 4: We can’t Override Parent Class final Methods in Child Class i.e., Overriding Concept is Not Applicable for final Methods

Case 5 : We can Override Parent Class Concrete Method as Abstract in Child Class.

• Next Level Child is the Responsible to Provide Implementation.

• The Main Advantage of this Approach is we can Stop Availability of Parent Class Method Implementation to the Next Level Child Classes.

Case 6: We can Override Parent Class Abstract Method in Child Class to Provide Implementation.

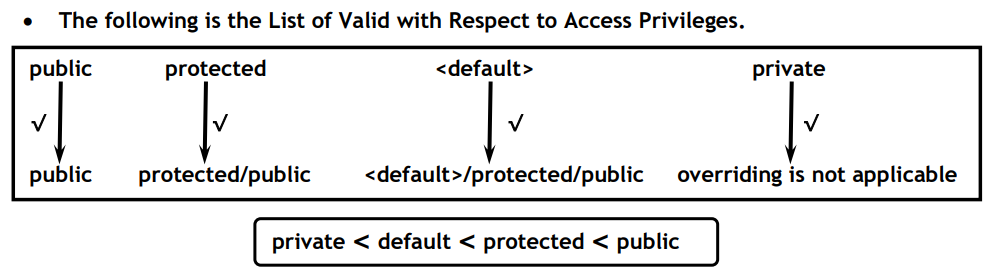
Case 7 : We can Override Parent Class Concrete Method as Abstract in Child Class.

Next Level Child is the Responsible to Provide Implementation.

• The Main Advantage of this Approach is we can Stop Availability of Parent Class Method Implementation to the Next Level Child Classes

Case 8: The Following Modifiers won’t Keep any Restrictions in Overriding. ♣ synchronized, ♣ native ♣ strictfp

Case 9:



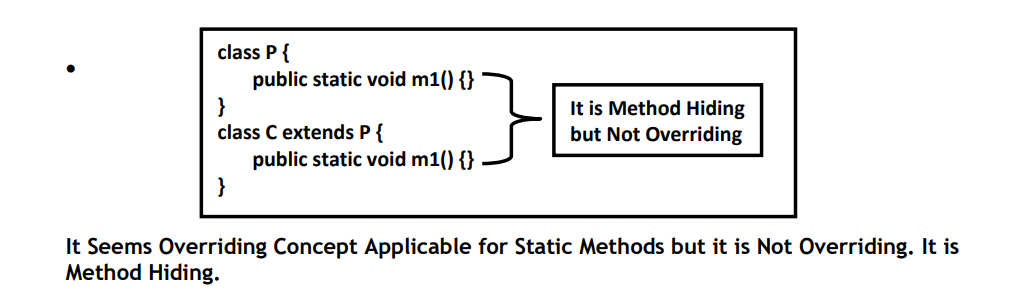
Note: we can’t reduce the scope of the access modifier in the overriding

Case 10: if the child class throw any checked Exception compulsory the parent class method should throw the same checked exception or it is parent.

Note: but no restriction on unchecked exception

Case 11: we can’t override a Static Method as Non-Static Otherwise will get CE.

Similarly, we can’t override a Non-Static Method as static.



A screenshot of a computer

Description automatically generated

Case 12:

Overriding concept is available only for method but not for variables.

Variables resolution always take care by compiler, based on reference type (but not based on Run Time Object) for rule is same for static and non-static member

**Exceptional Handling:**

Exception:- Unexpected event that happened that Disturb the Normal of the programme

Error: That can’t be handled and it is not programmer mistake but system issue.

Exception: That can be handled as it is due to programmer mistake.

What is purpose of Exception Handling?

Find an alternative way to continue the programme

A diagram of a diagram

Description automatically generated with medium confidence

**Runtime Stack Mechanism:**

For every thread stack is created and stack destroyed after execution by thread.

A computer code on a white background

Description automatically generated

A diagram of a computer

Description automatically generated

**Defaul Exception:**

• In Our Java Program Inside a Method if an Exception raised, then that Method is Responsible to **Create an Exception Object by including the following Information. ♣ Name of Exception ♣ Description of Exception ♣ Location of Exception (Stack Trace)**

• **After creating Exception Object Method Handovers that Object to the JVM.**

• JVM will Check whether Corresponding Method contain any Exception Handling Code OR Not.

• If the Method doesn't contain any Exception Handling Code then JVM Terminates that Method Abnormally and Removes Corresponding Entry from the Stack.

• JVM will Identify Caller Method and Check whether the Caller Method contain any Exception Handle Code OR Not.

• If Caller Method doesn't contain any Exception Handling Code then JVM Terminates that Caller Method and Removes Corresponding Entry from the Stack.

• This Process will be continued until main().

• If the main() also doesn't contain Exception Handling Code then JVM Terminates main() Abnormally and Removes Corresponding Entry from the Stack.

• Then JVM Handovers the Responsibility of Exception Handling to the Default Exception Handler, which is the Part of the JVM.

• Default Exception Handler Just Terminates the Program Abnormally and Prints Exception Information to the Console in the following Format.

Exception in thread: “Xxx” Name of the Exception: Description Stack Trace

**Note:**

• In Our Program if all Methods Terminated Normally, then Only the Program will be Terminated Normally.

• In Our Program if at least One Method terminates Abnormally then the Program Termination is Abnormal Termination.

**Exception Hierarchy**

• Throwable Class Acts as Root for Exception Hierarchy.

• Throwable Class contains 2 Child Classes Exception and Error

**Checked Vs Unchecked Exception:**

Checked – which is checked compiler whether we are handling or not

Unchecked - doesn’t checked by compiler

**Note:**

**Both exception occur only at runtime**

**TRY-CATCH Block**

**Note:**

**Try must be followed either catch or finally , we can’t use try alone**

**Otherwise Error: java: 'try' without 'catch', 'finally' or resource declarations**

**A white square with black text

Description automatically generated**

**Case 1:** If there is No Exception ◊ 1, 2, 3, 5, Normal Termination.

**Case 2:** If an Exception raised in Statement 2 and Corresponding catch Block Matched ◊ 1, 4, 5, Normal Termination.

**Case 3:** If an Exception raised at Statement 2 and Corresponding catch Block Not Matched ◊ 1 followed by Abnormal Termination.

**Case 4:** If an Exception raised at Statement-4 OR Statement-5 then it’s Always Abnormal Termination.

**Conclusions:**

• Within the try Block if any where an Exception raised then Rest of the try Block won't be executed even though we handled that Exception.

• Hence within the try Block we have to Take Only Risky Code and Hence Length of the try Block should be as Less as Possible.

• If there is any Statement which raises an Exception and it is Not Part of the try Block then it is Always Abnormal Termination.

• In Addition to try Block there May be a Chance of raising an Exception Inside catch and finally Blocks Also.

**Method to print Exception:**

**A white rectangular sign with black text

Description automatically generated**

**Note:**

1. Try must be followed by either catch or finally
2. In multiple catch block order must be child to parent
3. Finally always execute irrespective of whether exception is handled or not.
4. **In return - finally Block return Statement has More Priority than try and catch Block return Statements.**
5. It is ok it the finally don’t have return statement but try and catch must have return statement then.
6. Finally vs system.exit(0); finally wil not work

System.exit(0);

☀ We can Use this Method to Exit (Shut Down) the System (JVM) Programmatically.

☀ The Argument Represents as Status Code.

☀ Instead of 0 we can Pass any Valid int Value.

☀ 0 Means Normal Termination, Non- Zero Means Abnormal Termination

package com.DurgaNotes.Chapter6\_ExceptionHandling;  
  
public class Test

{  
 public static void main(String[] args) throws Throwable {  
 try  
 {  
 System.*out*.println(10/0);  
 }  
  
 catch (Exception e)  
 {  
 e.printStackTrace();  
 }  
  
 }  
  
}

Generally caller method is responsible to create a exception object and transfer it to the JVM

Here caller method is main so main method is responsible to create a exception object and transfer it to the jvm and jvm will handle it.

**Throw:**

When we use throw new ArithmeticException() and create exception object **to handover it to the JVM** manually we use **throw keyword.**

In General we can Use throw Key Word for Customized Exceptions but Not for pre-defined Exceptions.

Throw => to handover our created exception object to the JVM manually

**For unchecked exception use of try-catch is not necessary and compiler is not going to give any error.**

**And we don’t use throws for unchecked exception.**

**Throw and Throws keyword is only applicable to throwable types**

package com.DurgaNotes.Chapter6\_ExceptionHandling;  
  
import java.io.EOFException;  
import java.io.FileNotFoundException;  
import java.io.IOError;  
import java.io.IOException;  
  
  
class throwNewException extends IOException  
{}  
  
class throwNewException2 extends throwNewException  
{}  
  
  
public class practice\_ExceptionHandling  
{  
  
  
 public static void throwException2() throws IOException  
 {  
 *throwException*();  
 }  
 public static void throwException() throws IOException  
 {  
// throw new throwNewException();  
// throw new IOException();  
  
 throw new throwNewException2();  
  
 }  
  
 public static void main(String[] args) throws IOException  
 {  
  
// throwException2();  
  
 try {  
 System.*out*.println("Hello");  
 }  
 catch (ArithmeticException e)  
 {  
 e.printStackTrace();  
 }  
  
 }  
  
}

**Note: caller method must throw same or parent class exception**

**A close-up of a computer code

Description automatically generated**

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

Case 4: Inside try Block, if there is No Chance of raising an Exception then we can't write catch Block for that Exception. Otherwise we will get Compile Time Error Saying CE: exception XXX is never thrown in body of corresponding try statement. But this Rule is Applicable Only for Fully Checked Exceptions.

Case 5:

Overridden methods cannot throw new or broader **( parent )** checked exceptions than the one they inherit.

class PrintException extends Exception {}  
class PaperPrintException extends PrintException {}  
 interface Printer {  
 abstract int printData() throws PrintException;  
}  
  
public class Three implements Printer  
{  
  
 public static void main(String[] moat) throws Exception  
 {  
  
 }  
  
 @Override  
 public int printData() throws PaperPrintException {  
 return 0;  
 }  
}

Multi Catch Block (Catch Block with Multiple Exceptions)

Until 1.6 Version even though Multiple Exceptions having Same Handling Code Compulsory we have to write a Separate catch Block for Every Exception.

• The Problem in this Approach is it Increases Length of the Code and Reduces Readability.

• To Overcome this Problem SUN People Introduced Multi Catch Block in 1.7 Version.

• In this Approach we can write a Single Catch Block which can Handle Multiple Exceptions of different Types.

**• If Mutli Catch Block there should Not be any Relation between Exception Types (Like Parent to Child OR Child to Parent OR Same Type) Otherwise we will get Compile Time Error.**

Try

{ ------- }

catch (ArithmeticException | NullPointerException e)

{ e.printStackTrace(); }

catch (ClassCastException | IOException e)

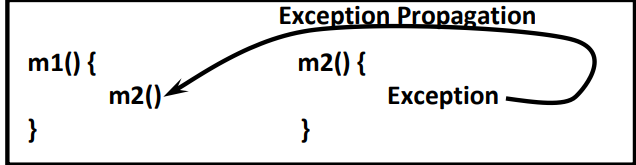
{ System.out.println(e.getMessage()); }

Exception Propagation:

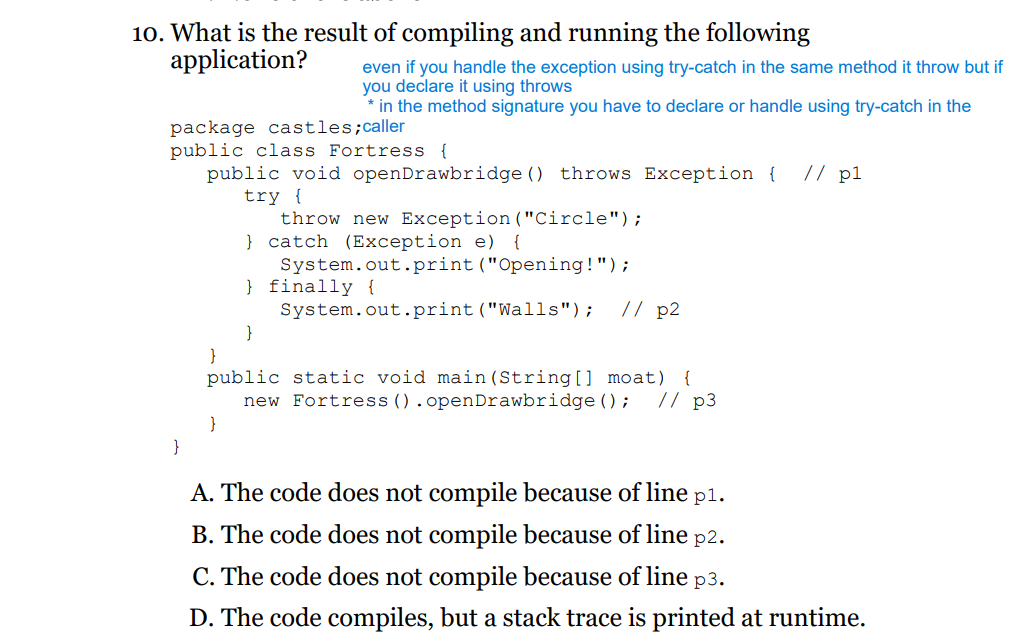
☀ Within a Method if an Exception raised and if we are Not Handle that Exception then that Exception Object will be propagated to Automatically to the Caller Method.

☀ Then Caller Method is Responsible to Handle that Exception.

☀ This Process is Called Exception Propagation



IMP Question :



A screenshot of a computer screen

Description automatically generated

String and StringBuilder

toString(); - of object class

* ClassName@Hexa\_Decimal\_String\_of\_hashcode

The hashCode()

Object class hashcode method generate hashcode based on Address of an object.

For overriding hashcode , make sure it generates different code every time

**toString() vs hashcode();**

toString method internally use hashcode.

whenever we are Overriding toString() Call hashCode()

A computer code with black text

Description automatically generated

A screenshot of a computer program

Description automatically generated

The equals():

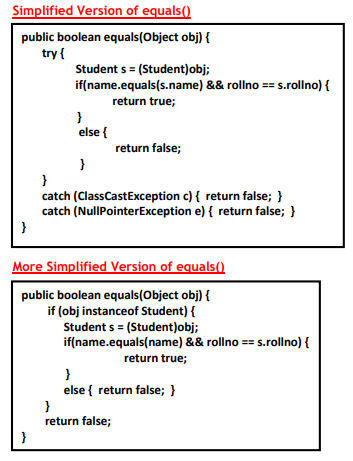
If our Class doesn’t contain equals() then Object Class equals() will be executed. Which is meant for Reference Comparison (Address Comparison) i.e. r1.equals(r2); Returns true if and only if both r1 and r2 pointing to the Same Object.

While Overriding equals() we have to Consider the following things.

• What is the Meaning of Content Comparison. For Example Whether we have to Check only Names OR only rollnos OR Both.

• If we Pass different Type of Objects our equals() should Return false but not java.lang.ClassCastException i.e Whether to Handle ClassCastException to Return false.

• If we Pass null Argument our equals() should Return false but not java.lang.NullPointerException i.e. we have to Handle NullPointerException to Return false.



**Relationship between ‘==’ Operator and .equals()**

• If r1 == r2 Return true then r1.equals(r2) is always true i.e if 2 Objects are Equal by == Operator then these Objects are always Equal by .equals() also.

• If r1 == r2 Returns false then r1.equals(r2) then we can’t conclude anything about equals(). It may Returns true OR false.

• If r1.equals(r2) is true then we can’t conclude anything about == Operator. It may Returns true OR false.

• If r1.equals(r2) is false then r1 == r2 is always false.

A screenshot of a computer

Description automatically generated

**Note:** In General we can use .equals() for Content Comparison whereas == Operator for Reference Comparison.

**Contract between .equals() and hashCode()**

• If 2 Objects are Equal by .equals() then Compulsory their hashCodes must be Same i.e. 2 Equivalent Objects should have Same hashCode. i.e. if r1.equals(r2) is true then r1.hashCode() == r2.hashCode() should be true.

• If 2 Objects are not Equal by .equals() then No Restriction on hashCodes may be Same OR may not be Same.

• If Hash Codes of 2 Objects are Equal we can’t conclude anything about .equals() it may Returns true OR false.

• If Hash Codes of 2 Objects are not Equal then these Objects are always not Equal by .equals() Also.

We have same relation of

**== and .equals()**

**.equals() and hashcode();**

**Difference between String and StringBuffer Objects?**

**A computer screen shot of a computer

Description automatically generated**

**Note:**

* + - StringBuffer .equals() is not overridden for content comparison
    - String is mutable and StringBuffer is not mutable
    - New String() create object in SCP (String Constant Pool) and heap
    - String create object only in SCP if it is not there and also not applicable for Garbage Collection
    - Main Advantage of SCP is Memory Utilization and Performance will be improved.

**In Addition to String Objects any Other Objects are Immutable in Java?**

**All Wrapper Class Objects also Immutable.**

A diagram of a spring

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated

A diagram of a software company

Description automatically generated

A screenshot of a computer

Description automatically generated

final Vs Immutability:

• final Keyword applicable for Variables but not for Objects whereas Immutability Concept is applicable for Objects but not for Variables.

• final and Immutability both are different Concepts.

• By declaring a Reference Variable as final we won’t get any Immutability Nature but we can’t perform re-assignment for that Reference Variable.

A white background with black text

Description automatically generated

A white background with black text

Description automatically generated

StringBuffer – Thread Safe

StringBuilder – not Thread Safe

**COLLECTIONS:**

class Three  
{  
  
 public static void main(String[] args)  
 {  
  
// Collections Method  
// 1) boolean add(Object o)  
// 2) boolean addAll(Collection c)  
// 3) boolean remove(Object o)  
// 4) boolean removeAll(Collection c)  
// 5) boolean retainAll(Collection c): To Remove All Objects Except those Present in c.  
// 6) void clear()  
// 7) boolean contains(Object o)  
// 8) boolean containsAll(Collection c)  
// 9) boolean isEmpty()  
// 10)int size()  
// 11)Object[] toArray()  
// 12)Iterator iterator()  
  
  
// Constructor  
// ArrayList l = new ArrayList();  
//--> New Capacity = (Current Capacity \* 3/2) + 1  
//  
// ArrayList l = new ArrayList(int initialCapacity);  
//  
//  
// To Provide Support for this Requirement Every Collection Class Implements  
// Serializable and Cloneable Interfaces.  
//  
// ArrayList and Vector Classes Implements RandomAccess Interface. So that we can  
// Access any Random Element with the Same Speed.  
  
// RandomAccess Interface Present in java.util Package and it doesn't contain any  
// Methods. Hence it is a Marker Interface.  
  
  
  
 ArrayList<String> a = new ArrayList<>();  
  
 a.add("relam 0");  
 a.add("relam 1");  
 a.add("relam 2");  
 a.add("relam 3");  
  
 ArrayList<String> b = new ArrayList<>();  
  
 b.add("b 0");  
 b.add("b 1");  
 b.add("b 2");  
 b.add("b 3");  
// System.out.println(b);  
  
 a.addAll(b);  
// a.remove("relam 0");  
// a.removeAll(b);  
// a.retainAll(b);  
// a.clear();  
  
// System.out.println(a.containsAll(b)); // true  
// System.out.println(a.contains("relam 0")); // true  
  
 System.*out*.println(a.add("relam 0"));  
// System.out.println(a);  
  
  
  
 ArrayList<String> c = new ArrayList<>(b);  
 System.*out*.println(c); // [b0, b1, b2, b3]  
  
 ArrayList d = new ArrayList<>(b);  
 Three g = new Three();  
 d.add(g); // instead of g if we add new Three(), it will consider it as a new object and will not give result.  
 System.*out*.println(d.indexOf(g));  
  
  
 ArrayList<Integer> l = new ArrayList<>();  
 l.add(1);  
 l.add(2);  
 l.add(3);  
 l.add(4);  
 l.add(null);  
 l.remove(2);  
 l.remove(null);  
 System.*out*.println(l);  
  
  
  
  
 }  
  
}

**Wrapper Class**

package com.OCJA.Practise;  
  
import java.lang.reflect.Constructor;  
import java.time.\*;  
import java.time.format.DateTimeFormatter;  
import java.util.ArrayList;  
  
class Three  
{  
  
 public static void main(String[] args)  
 {  
 /\*  
  
 Wrapper Class - to convert primitive into object  
 or  
 to handle primitive in the way we can handle objects  
  
 Constructor  
  
 <T> <variable> = new <T> (<T>);  
 <T> <variable> = new <T> (<String>);  
  
 String, primitive --> Wrapper Object - valueOf();  
 Wrapper Object --> primitive - xxxvalue();  
 String --> primitive - parsexxx();  
  
 IMP - boolean, integer and double  
\*/  
  
// Integer  
  
 Integer a = new Integer(7);  
 Integer b = new Integer("7");  
  
  
 // for boolean value can be booelan or string but must be true or false  
 // even capital letter are acceptable.  
  
// Boolean  
  
 Boolean c = new Boolean(true); // true  
 Boolean d = new Boolean("false"); // false  
  
 Boolean e = new Boolean("FALSE"); // false  
 Boolean e1 = new Boolean("TRUE"); // true  
 Boolean f = new Boolean("left"); // false  
  
 System.*out*.println(a+b);  
 System.*out*.println(c);  
 System.*out*.println(d);  
 System.*out*.println(e);  
 System.*out*.println(f);  
  
// Double  
  
 Double g = new Double(20.3);  
 Double h = new Double("20.3");  
 Double i = new Double(20.3f);  
 Double j = new Double("20.3f");  
  
 System.*out*.println(g);  
 System.*out*.println(h);  
 System.*out*.println(i);  
 System.*out*.println(j);  
  
  
 Integer aa = Integer.*valueOf*("1014");  
 Integer a1 = Integer.*valueOf*(2);  
  
 int ap = a1.intValue();  
 int ap2 = Integer.*parseInt*("20");  
 System.*out*.println(ap);  
 System.*out*.println(ap2);  
  
  
  
  
  
  
  
  
 }  
  
}

**Date and Time API**

\*\*\***LocalDate and LocalDateTime are immutable**\*\*

package com.OCJA.Practise;  
  
import java.time.\*;  
import java.time.format.DateTimeFormatter;  
  
class Three  
{  
  
 public static void main(String[] args)  
 {  
 LocalDate s = LocalDate.*now*();  
 s.plusDays(30);

LocalDate a = LocalDate.*of*(1993,12,07);

LocalDateTime ldt = LocalDateTime.*now*();  
 LocalDateTime b = LocalDateTime.*of*(1993,12,07,12,12,12);

// ZoneId z = ZoneId.of("America/Los\_Angeles");  
// ZonedDateTime zdt = ZonedDateTime.now(z);

ZoneId z = ZoneId.*systemDefault*();  
 ZonedDateTime zdt = ZonedDateTime.*now*(z);  
  
 Period p = Period.*between*(s, s.plusDays(30));  
  
// System.out.println(p);  
 System.*out*.println(zdt);  
// System.out.println(s);  
  
  
 LocalDate ldp = LocalDate.*parse*("1993-12-23");  
 System.*out*.println(ldp);  
  
 String formatdate = ldp.format(DateTimeFormatter.*ISO\_DATE*);  
// System.out.println(formatdate.length());  
  
  
// both are method of localDate  
// parse - > string to localdate  
// format -> localdate to string.  
  
  
 LocalDate Class Parse Method;  
 public static LocalDate parse(CharSequence text);  
 public static LocalDate parse(CharSequence text, DateTimeFormatter formatter);  
 public String format(DateTimeFormatter formatter);  
  
 Note: LocalDate,LocalTime and LocalDateTime are immutable ojects  
  
  
 }  
  
}

**Predicates and Lambda:**

interface Predicate<T>

{

public boolean test(T t);

}

Predicate<String> a = e -> e.length()>5;  
Predicate<String> b = e -> e.length()<20;  
System.*out*.println(a.and(b).test("Aniruddha"));  
  
lambee s = d -> d.length();  
  
System.*out*.println(s.learn("Aniruddha"));