LowLevelLanguage/MachineLevelLanguage- Machine can undestand only binary language. It is in binary format. In the form of 0's and 1's

AssemblyLanguage/MidLevelLanguage- Intermediate language which looks like processsor instructions/cmds which is not completely in binary

and also not completely human readable.

HighLevelLanguages - Human Readable or more like general english.

int a=5;

int b=10;

int result = a+b;

Compiler : Compiler is a program that converts human-readable code to machine-readable instructions.

eg for compiled languages C, C++, Objective-C, Scala

Interpreter : Interpreter convert the humanreadable code to machine code while executing the program line by line.

eg for interpreted languages:

Python,Ruby, JavaScript,

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

COMPILED LANGUAGES

are written in a code that can be executed directly on a computer’s processor. This is because a compiler has translated the code into the computer’s “native” language up front, well before the program is even run. This process can take many passes before it is optimized as machine code, but the output is always code that’s ready to be executed—and that executes efficiently, as a result.

A compiler is a program that converts human-readable code into computer-readable instructions—a process that only happens once in the lifespan of that code. Initially, it takes a bit longer because the compiler has to rearrange, optimize, or “compile” object code first.

Once a compiler takes in source code, optimizes it, and then generates the object code, another step has to happen. Object code can’t always run on its own; it needs dynamic load libraries and other bits of code—code that’s unique to the operating system, which is housed in those libraries. Within the compiling process, these libraries are linked to object code in a “linker,” a part of the compiler that bundles those bits of code together, then it’s good to go.

That final product—the object code packaged up with the libraries—is the compiling program’s executable file—a .exe file.

Some compiled languages include:

• C

• C++

• C#

• Erlang

• Objective-C

• Pascal

• Scala

• Swift

• Smalltalk

• TypeScript

INTERPRETED LANGUAGES

An interpreted language is any programming language that isn’t already in “machine code” prior to runtime. Unlike compiled languages, an interpreted language’s translation doesn’t happen beforehand. Translation occurs at the same time as the program is being executed.

Many of an interpreted language’s instructions can be executed directly, without compiling to machine code; however, when certain code is required, an interpreter steps in during runtime and translates it on the spot.

Interpreted languages have a major advantage: they’re portable, which means they can run on different operating systems and platforms. Also, because they’re translated on the spot, they’re going to be optimized for the system on which they’re being run. That means there are no middle steps, less memory space is required for interim object code, and there’s no need to worry about platform-specific code.

How interpreting works

The interpreter is a program that converts source code—the human-readable code mentioned above—into machine code each time you run the program, one line at a time. It starts interpreting each instruction immediately upon execution, which means that the resulting program runs slower than a compiled program—it’s got more going on at runtime. Compiled languages, on the other hand, have already been through this translation before program execution, so they’re arguably faster.

Interpreters have some other bonuses, too. They’re especially helpful for reviewing, running, and testing an application’s functionality during development because they’re able to execute high-level programs immediately—and generate helpful error reports. Also, they allow programmers to make small, step-by-step changes during the development process, incrementally, which complements a step-by-step process for adding and then testing smaller sections of an application.

Some interpreted languages include:

• Java

• JavaScript

• PHP

• Perl

• Python

• Ruby

Simula was the first object-oriented programming language. Java, Python, C++, Visual Basic .NET and Ruby are the most popular OOP languages today. The Java programming language is designed especially for use in distributed applications on corporate networks and the Internet. Ruby is used in many Web applications. Curl, Smalltalk, Delphi and Eiffel are also examples of object-oriented programming languages.

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

Java- compiled and interpreted.

Java is an Object Oriented programming lanaguage.

It has Classes and Objects.

Mobile class - Class defines model or blueprint for object. It has properties and methods.

Object is that which exist or it is instance of class.It has state(properties) and behaviour(action/methods in which we define logic).

Samsung, Apple, GoogleNexus - has color, manufacturer, ram features, camera.

Object Oriented Programming Concepts / OOPS:

Inheritance

Polymorphism

Abstraction

Encapsulation

Java - It is platform independent language.Write Once Run Anywhre (WORA).

Java is a language and platform.

I have written a program in java and compiled it in windows OS, I can still execute this program in MAC OS.

Features of Java:

Simple- It eliminates pointers which are used in c and c++ for memory allocation and deallocation. Java memeory managemenr is taken care by JVM.

ObjectOriented- In Java everything is objects and classes

Distributed- It works with various systems on n/w and can handle any protocols(Tcp/ip,udp etc)

Robust- It is strong bcz it do not crash easily..Java handles error/exception in program using exception handling features.

Secure- It supports encryption and decryption mechanisms.

SystemIndependent- Java byte code can be executed on any processor and OS.

Portable

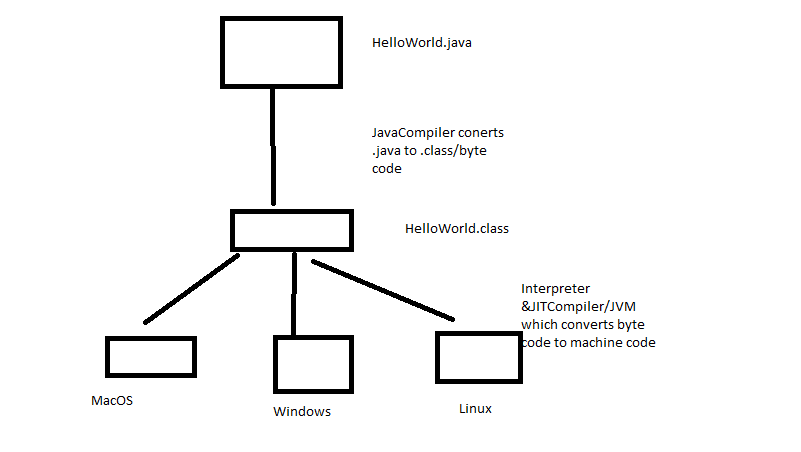
Interpreted- Hello.java--> Hello.class(bytecode)-->machine readable code is done by interpreter

Good Performance- Bcz it handles complex logics execution using JIT compiler - JustInTimeCompiler

MultiThreaded- Java supports threads which help logic to execute parallely

Scalabilty- It can connect to DB, webservices and any remote calls using its API packages.

Dynamic- Make dynamic interactions possible on web apps.



Setup java:

Download Java as per your OS from - http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Run exe file. You wil Java folder in C:ProgramFiles

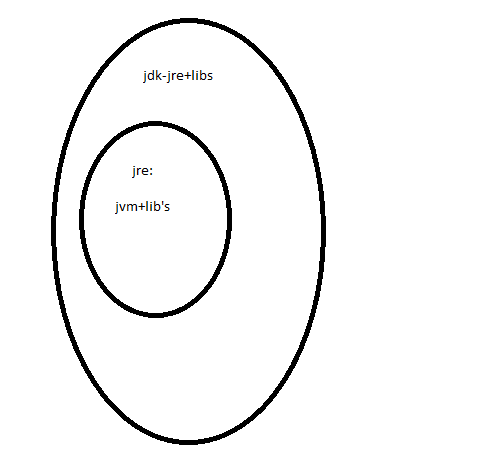
Setup Environment Variables - Add new variable - JAVA\_HOME - C:\Program Files (x86)\Java\jdk1.8.0\_101

and path variable edit and append- %JAVA\_HOME%\bin or C:\Program Files (x86)\Java\jdk1.8.0\_101\bin

JDK - Java developement Kit - set of Java Libraries and JRE - required to develop a program

JRE - Java Runtime Environment - Few libraries (Which has subset of certain libraries required to execute the program) and JVM - executes program

JVM - JavaVirtualMachine- Interprets the program



Java is system independent but jvm is system dependent.

GarbageCollection- memory management section in jvm - this will deallocate memory after program execution.

dll and exe:

dll- dynamic link mibrary- which are used by exe file in their execution

exe- they are executable on machine.

JavaVersions-1 to 1.4, 1.5, 1.6, 1.7, 1.8

|  |  |
| --- | --- |
| **Release** | **Year** |
| JDK Beta | 1994 |
| JDK 1.0 | 1996 |
| JDK 1.1 | 1997 |
| J2SE 1.2 | 1998 |
| J2SE 1.3 | 2000 |
| J2SE 1.4 | 2002 |
| J2SE 5.0 | 2005 |
| Java SE 6 | 2006 |
| Java SE 7 | 2011 |
| Java SE 8 | 2014 |

IDE's - eclipse, intellijidea, NetBeans - Integrated dev environment

**Package** – package is the namespace/folder structure that organizes the set of classes and interfaces.

**Class** : Model or blueprint which has properties and methods.

Members of class are variables and methods.

**Variables** are properties which define state of object:

Syntax- accesModifier Datatype varName ;

**public** String breed;

Variables at class level are called global variables or instance variables and they can be referred in any of the methods of class.

**Methods** are actions which define logic of object.

Syntax- accesModifier returnType methodName(input parameters){

Code or method logic

}

**public** **void** eat(){

System.***out***.println("It eats food");

}

**Object**- instance/representation of class.This has properties and it can call class methods.

Syntax- ClassName objName = new ClassName();

Animal animalObj = **new** Animal();

**NamingConventions:**

Java is caseSensitive.

packageNames – lower case

com.wbl.java

com.wbl.test

variableNames, methodNames – camelCase

display

displayDetails

classNames, interfaceNames – initCaps

Student

StudentDetails

Constants- allcaps

PI=3.14

**Rules for Identifiers/Names:**

Identifiers can have alphabets, numbers, $,\_

Do not start name with number, it can start with alphabets,$ or \_ and it can include numbers.

Each identifier must have atleast one character.

**AccessModifiers:**

Public – can be accessed from anywhere.

Private- can be accessed only **within the class**

Protected- can be accessed within same package **and subclass of diff package.**

Default- can be accessed within the **same package.**

**JavaComments:**

//- single line comments

/\*…\*/- multi line comments

**Constructor in java:**

Constructor is similar to method syntax wise but it is called only **at the time of creation of object.We can make any intialisations using constructions.**

It wil have same name as class name.

It will not have return type.

Its added by default by java if you don’t create one.

It can take inout parameters like method.

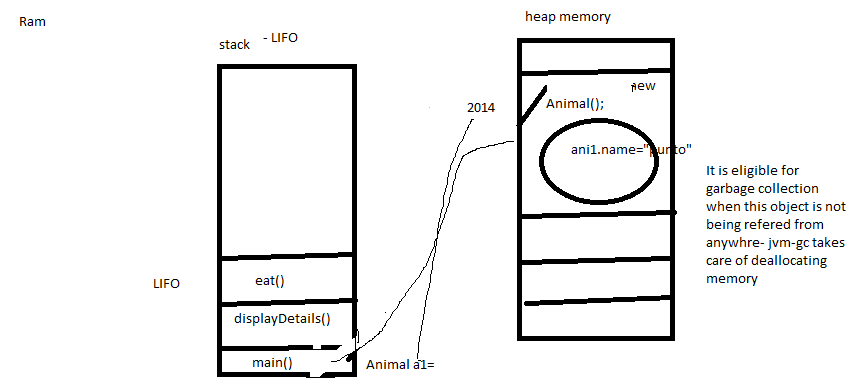
A class can have multiple constructors which is called constructor overloading.

Overloading means you will have same constructor name and different parameters within same class.

**Stack and heap in Java:**

Method execution are done from stack- stack follows LIFO principle.

Objects memory allocation is done in heap

****

primitiveDatatypes:

keywords in java:

this

static

final

Assignment-1:

Create a class and define properties and methods.

Create Objects for class and access properties and methods.

**OOPS**:

**Encapsulation**: Binding the variables and methods together , hide implementation behind an interface.

Encapsulated code will have 2 things:

Instance variables are protected(we can use private modifier and protect it)

We provide public getters and setters to access the instance variables.

**Inheritance**:

Inheritance allows a class to be subclass(child class)of a superclass(parent class) – therefore subclasses inherit public and protected variables and methods of the superclass.

Use: Avoid Repeating the logic or redundancy of code – Making the code reusable.

Car- Benz is a car, BMW is a car

Car{

Public void doARide(){

Print(“evry car can do a ride”);

}

Benz extends Car(){

Public void setBenzFeatures(){

Print(“luxury car”);

}

}

BMW extends Car{

}

Polo extends WV{

}

Animal – Dog is a animal, Cat is animal

Class Animal{

Public void commonFeatures(){

Print(it has four legs)

}

Public void setSound(){

Print(“they make sounds”)

}

}

Dog extends Animal{

Public void setSound(){

Print(Dog says bow)}

}

Cat extends Animal{

Public void setSound(){

Print(Cat says meow)

}

}

**MethodOverriding: Overriding only occurs with inheritance.**

The child class methods should have exactly same syntax as parent class - same name, same return type.

You cannot have more restrictive access modifier or you cannot narrow the scope.

You can have less restrictive access modifier or widen the scope.

Lifet🡪right – narrow-wide or morerestrictive->lessRestrictive

Private->default->protected->public

Exceptions

Child class should have same return type as parent but Covariant return type is allowed.

Final methods cannot be overridden.

static methods cannot be overridden.

A parent class reference can hold child class objects.

Eg: all below are valid

Animal dog = new Dog();

Animal cat = new Cat();

Dog dog = new Dog();

Cat cat = new Cat();

In case of overriding - the call of the method whether it should call parent class method or child class method depends on which object is getting created at run time.This is called **java virtual method invocation.**

**super:** is a keyword by which we can access parent class members(methods,variables,constructors) in child class.

It is used when you want to refer parent class methods/variables in child

When both parent and child have similar variables/methods.

**this:**  keyword in java – this is used to refer the member(instance variables,methods) of current object from method or constructor.

It can be used inside a method or a constructor of the class.

this is a reference to current object of execution.

**Abstraction:**

Annotations – that which give certain functionality in easier way

Assignment-2:

Write a class that describes few fields and methods w.r.to Training.

Write subclasses for QA and UI training with specific implementation for recordings and

course content.

**TypeCasting:**

ImplicitCasting / downcasting : Assigning subclass reference which holds subclass object to superclass reference.

ExplicitCasting/upcasting: Assigning super class refernce which holds subclass object to subclass reference.

We cannot assign super class reference which holds super class object to a subclass refrence.It do not throw any compile time erroe but at run time it throws class cast exception.

**Polymorphism** – exisiting in many forms….

A singe reference variables can take its own type of object and also its subtype object – through inheritance

A single method can be given different behaviors (different logics) by method overriding – through inheritance

Strictly speaking – polymorphism is applicable to overriding but not overloading.

Polymorphism: Existence of same thing in multiple forms.

Animal a1 = new Animal();

Animal a2 = new Dog();

a2.makeNoise();

Animal a3 = new Cat();

Dog d1 = new Dog();

Cat c1 = new Cat();

c1.makeNoise();

If you have makeNoise method in both animal and dog class,

At run time the object type decides which method is invoked, at compile time

Reference type decides which method is called.

Polymorphism is applicable w.r.to overriding same method exist in both sub class and super class

And we call any of these methods based on the object created.

**MethodOverloading**: Allows to have multiple methods within same class with same method name.

You can have same method name but different method parameters.

Either the count of the input parameters should be modified or the dayatype of the input parameters should be modfified.

You may or may not have same return type.

You may or may not have same access modifier.

The actual method that’s invoked is still a virtual method invocation that happens at runtime, but the compiler will already know the signature of the method to be invoked

You can throw a new or broader exceptions.

|  |  |
| --- | --- |
| Overloading | Overriding |
| Method arguments **must** change-either by number of args or by datatype | Method arguments **must not** change |
| Return type can be changed | Cannot change the return type except the covariant return types |
| Exceptions declared in method signature can change | Cannot change exceptions of super class in subclass method overriding-u can still add any unchecked (runtime)exceptions,narrower checked exxcpetions, but u cannot throw new or broader checked exceptions |
| Access modifiers can change | Cannot make **more restrictive** acces modifier or we cannot reduce the visibility… |
| It decides which method to call at compile time- it exhibits virtual method invocation at run time | At run time based on which object is getting(subclass or superclass) created it calls that particular method, but at compile time it just decides on object reference – so we need to make sure that the reference objet has the method u r calling at compile time. But remember that at runtime, Java uses virtual method invocation to dynamically select the actual version of the method that will run, based on the actual instance |

Overloading: static(compile time) polymorphism

at compile time only depending on method arguments(for overloading) it decides which method to call

The actual method that’s invoked is still a virtual method invocation that happens at runtime, but the compiler will already know the signature of the method to be invoked

Overriding – dynamic(runtime)polymorphism - at compile time your object referenece decides the method call , at run time - actual object decides which method to call

But remember that at runtime, Java uses virtual method invocation to dynamically select the actual version of the method that will run, based on the actual instance

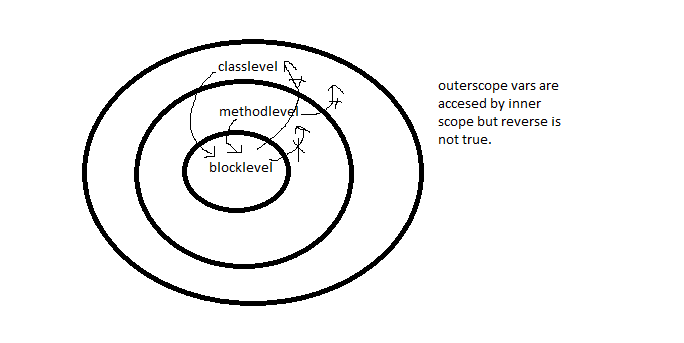
**Different types and scopes of Variables in Java Class:**

Global variables/Instance variables – variables out side the method and within the class – accessed anywhere in class.

Method variables /local variables– which are inside the method-– accessed within method

Method parameters- accessed within methods

Block variables - Variable inside Block --– accessed within blocks



**Static**: keyword in java – -class level but not object/instance level.

We can access static methods and variables by class name directly, we do not need object to be created

to access static methods/ variables.

The variable data gets shared between the objects if it is static. If it is non static every object

Has its own copy of instance variables.

If its static every class has its own copy of data.

We do not need to create objects to access these variables unlike instance variables/ methods..

We cannot acces non static variables /methods inside a static method.

Why because – static variables do not need objects and they are loaded at the time your class gets loaded—they don’t need objects but instance var’s need objects..

**Final:** Cannot modify the value once assigned.

Final is applicable to variables, methods and classes

If variable is final- you cannot reassign the data.

If method is final – you cannot override the method in subclass.

If class is final – you cannot create a subclass for that class.

Assignment-3:

Write a program to explain method overloading

Write a program to explain method overriding

Write a program that has a method which return the count of objects created for that class.

Write a method which can be called without creating the object from outside the class and have some variables in class which should not be modified by anyone.

**AbstractClass:**

Abstract class in java is declared with keyword abstract, which can have both abstract methods and concrete methods.

Methods with method body/implementation are called concrete methods.

Methods without methodbody/ unimplemented are abstract methods.

We cannot create object for Abstract class.

It can have constructor and it is called when you are creating the child class object.

It allows to define public/private/protected/default scope for variables or methods.

If child class do not implement any abstract method then child class should also be declared abstract.

An abstract class can extend another abstract class.

**WhenToUse:**

Whenever we want to provide common logic to be used by child classes and also we want to

Set some abstract methods to be implemented by child(like setting rules for child class).

Whenever we want to provide common logic and if you think that creating object for this class is meaningless until it has further implementations or just to say that this class is incomplete.

Whenever you want use common variables with different values to be set by child classes and to have any(public/private/protected/default) scope for variables.

**Interface:**

Interface is 100% abstract class because it has only abstract methods.

Interface is a contract/setting rules for child classes.

We cannot add constructor for interface and we cannot create object for interface.

By default all methods are abstract in interface and it allows public and default scopes for methods.

By default all variables in interface are public, static and final i.e., they act as constants.

It supports multiple inheritance.

A class can extend other class and also implement interface at same time.

**WhenToUse**:

Whenever you do not have any common logic but we just want to set rules/contract for child classes by

declaring abstract methods.

When you are sure that you may need to add few more methods to implement in future, and you don’t want to disturb all existing child classes.

Eg:

Assume you have 100 child classes for interface and u want add new methods after few days and then all 100 classes will be forced to implement new feature if u add in existing interface.

If you don’t want to disturb existing things u can add it in a new interface and child classes which require that method will implement by multiple inheritance.

Also when multiple inheritance is needed we can use interface because in java we cannot achieve multiple inheritance through classes.

**MultiLevelInheritance** – Java supports multi level inheritance for classes and interfaces.

Class A

Class B extends A

Class C extends B

**MultipleInheritance**- Java supports multiple inheritance for interfaces.

interface A- test()

interface B-test()

Class C implements A,B

Eg: Class A{

Void abc(){

}

}

Class B extends A{

Void abc(){

}

}

Class C extends A{

Void abc(){

}

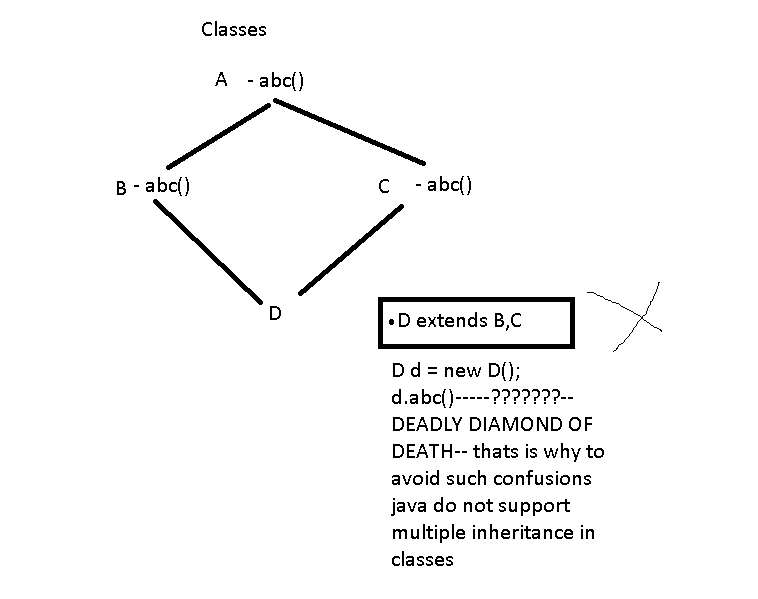
}

Class D extends B,C {

Void abc(){

}????????????whether is hud refer B- abc method or C -abc method

}



**Primitive Data Types**: Java is statically typed language.

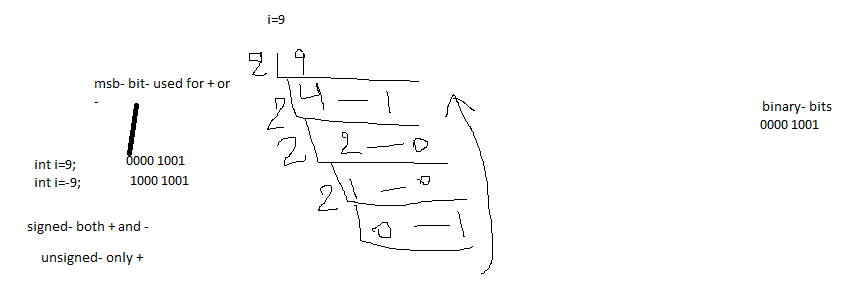
int id=1;// integer number

float price=56.7;// decimal number

String greet =“hello”;// text data

There 8 primitive data types in Java, primitive means it’s not class or object in java, it’s just keyword.

Other than these 8 data types everything is an Object type in Java.



Byte – 8 bit integer -128 to 128

Short- 16 bits integer

char – 16 bit Unicode character- any keyboard character - ‘a’

int- 32 bits integer

long- 64 bits integer

float- 32 bit decimal

double- 64 bit decimal

boolean – true or false

String is class in java- its not primitive datatype- it is set of characters or it is character array.

method var should be intialised before using the variable in method

Global variables get default values if you don’t initialize.

Primitive datatypes there are predefined default values.:

|  |  |
| --- | --- |
| **Data Type** | **Default Value (for fields)** |
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0L |
| float | 0.0f |
| double | 0.0d |
| char | '\u0000' |
| String (or any object) | null |
| boolean | false |

For Objects the default is null;

Assignment 4:

Create abstract class and implementations, interface and implementations

By taking any real world example like Mobile, System, Car, Bike etc.

**TypeCasting**:

**TypeCasting with Objects:**

TypeCasting is mainly for refernce variables w.r.to JavaClasses.

1. Child reference can always go to parent refernce – implicit or upcasting
2. Parent reference can go to child refrence only if that parent ref is holding child object- also here

It needs explicit casting or downcasting

3.parent reference to child refernce where parent ref is holding parent object wil not give any compilation issue provided you so explicit casting but gives runtime exception- classcast exception.

//TypeCasting

Mobile m1 = **new** Mobile();

AndroidMobile a1 = **new** AndroidMobile();

m1 = a1;// implicit casting or upcasting- child to parent

Mobile m2 = **new** AndroidMobile();

AndroidMobile a2 = **new** AndroidMobile();

m2=a2;// implicit casting or upcasting- child to parent

//AndrioidMobile is a Mobile

a2=(AndroidMobile)m2;// explicit casting or downcasting- parent to child

Mobile m3 = **new** Mobile();

AndroidMobile a3 = **new** AndroidMobile();

m3=a3;// implicit casting or upcasting- child to parent

//Mobile is a AndroidMobile- wrong , Mobile can IOs or windows or any

a3=(AndroidMobile)m3;//this is wrong bcz downcasting is done only when object satisfies is a relation

**TypeCasting with primitive datatypes:**

Left to right – any datatype from left can be assigned to right without any eplicit casting

Left->right-implicit casting or upcasting

byte-> short->int->long->float->double

right->left – it is explicit casting or downcasting and right to left is possible only if ranges are satisfied

**Operators:**

**Simple Assignment Operator**

= Simple assignment operator

**Arithmetic Operators**

+ Additive operator (also used

for String concatenation)

- Subtraction operator

\* Multiplication operator

/ Division operator

% Remainder operator

**Unary Operators**

+ Unary plus operator; indicates

positive value (numbers are

positive without this, however)

- Unary minus operator; negates

an expression

++ Increment operator; increments

a value by 1

-- Decrement operator; decrements

a value by 1

! Logical complement operator;

inverts the value of a boolean

**Equality and Relational Operators**

== Equal to

!= Not equal to

> Greater than

>= Greater than or equal to

< Less than

<= Less than or equal to

**Conditional Operators**

&& Conditional-AND

|| Conditional-OR

?: Ternary (shorthand for

if-then-else statement)

**Type Comparison Operator**

instanceof Compares an object to

a specified type

**Bitwise and Bit Shift Operators**

~ Unary bitwise complement

<< Signed left shift

>> Signed right shift

>>> Unsigned right shift

& Bitwise AND

^ Bitwise exclusive OR

| Bitwise inclusive OR

Some Uses of bitwise operator:

<http://www.leepoint.net/data/expressions/bitops.html>

Operators Precedence:

<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html>

Assignment 5:

write a program to practice primitive typecasting ?

write a program to practice Object refrence typecasting ?

write a program to practice operators on primitive datatypes?

**ConditionalStatements:**

**If.. else.. – can be used for single if else conditions**

**If..else if..else - can be used for multiple if else conditions**

**Switch..case**

**switch(param){**

**case condnString :**

**…..**

**default:**

**}**

**The param in switch block is allowed to have below datatypes:**

**byte, char, int, short, enum, String, Byte, Character, Integer, Short**

**Loops:**

**while – executes the block of code inside while until the condition is true.**

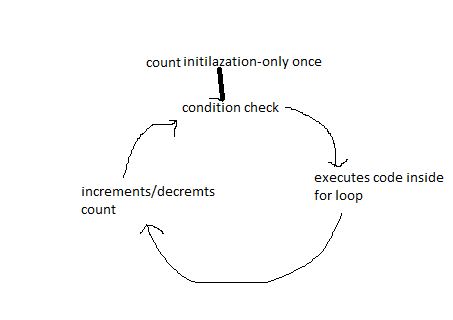
**do..while - executes the block of code inside while until the condition is true. But it executes the do block atleast once even when while condition do not match.**

**for loop**

**for( intialization of loop variable ; condition till where the loop should execute ; incremet/decrement){**

**--logic---**

**}**

****

**Basic Array and Collections iterations- we use for loops frequently.**

**Whenever we are not sure about end parameter for condition(eg: db result set) we use while over for loop bcz while keeps continuing loop until condition is true.**

**Branching statements:**

**Break – break wil come out of loop without executing any of further statements and iterations of loop.**

**Continue- continue will come out of current iteration without executing any further loop code for this iteration and proceed with next iteration.**

**Arrays – which is used to store similar kind of data(primitives and objects) at one place. Arrays is an object in java.**

**float price = 29.20f;**

**float[] prices = {29.20f, 56.6f, 78.9f, 44.5f}**

**int[] sids = {1,2,3,4,5,6,7,8,9,10}**

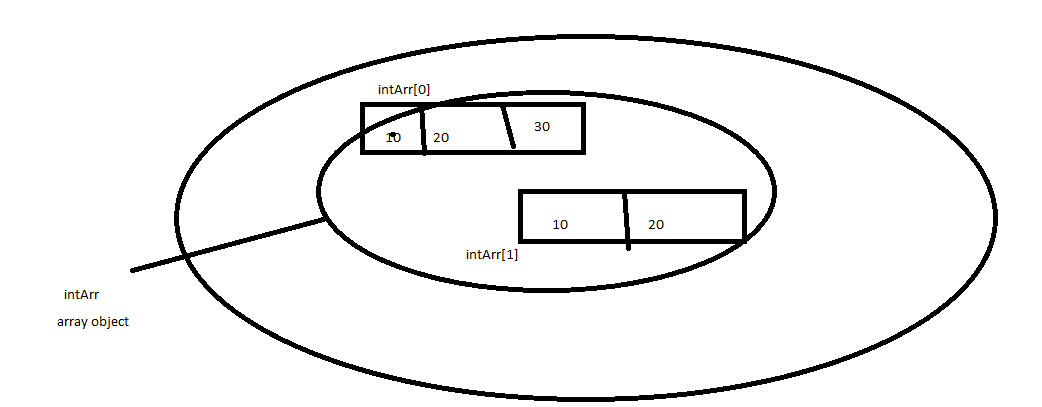
**String[] snames = {“geetha”,”savita”,”venna”}**

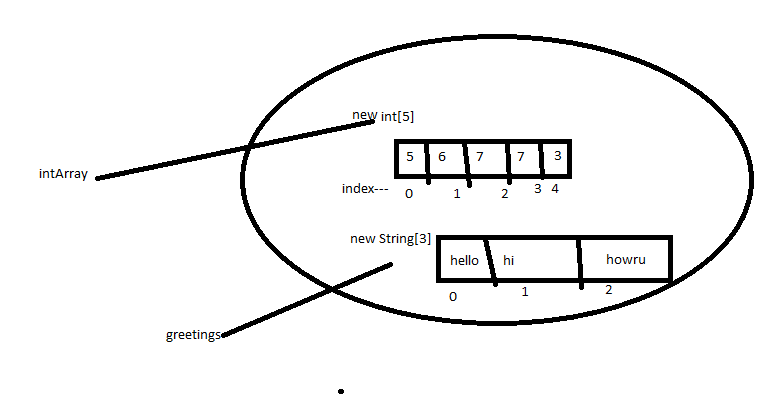
**Employee e1 = new Employee(1, “swetha”);**

**Employee e2 = new Employee(1, “swetha”);**

**Employee e3 = new Employee(1, “swetha”);**

**Employee[] emps = {e1,e2,e3};**





**Assignment 6:**

**Display all even numbers until given input n starting from 1.**

**Print multiples of 7 between 1 and 100.**

**Write a program to declare, intialise and iterate array data in all possible ways- for single dimensional and 2 dimensional.**

**String Class in Java:**

String is a class in java.lang package , it is under Object hierarchy like all other classes in java.

**String is immutable in Java means we cannot modify the objects once created.**

String is set of 16 bit unicode characters….

**We can create String in 2 ways:**

1. **Using new keyword- which creates string in heap memory**
2. **Using String literals- which creates string in StringConstantPool**

**StringConstantPool- It is used to save memory as String is most widely used in any application.**

**How it works:**

**If there is a string object which is already existing in pool and someone is trying to create the same value again-the new reference is pointed to existing object in pool rather than creating a new object again.**

**Eg: String s1= “hello”**

**String s2=”hello”**

**No of Objects created in pool - 1**

**Heap- Even though same object already exist- whenever new keyword is used it creates object again.**

**Eg: String s1= new String(“hello”)**

**String s2=new String(”hello”)**

**No of Objects created in heap – 2**

**Each time we try to modify existing String object- a new object gets created in memory but the old one is not modified. We can make the old reference point to newly created object. This immutable concept is applicable for any way- heap or pool.**

**String s = new String(hello);**

**s.concat(“world”); // new object is created**

**Why string is immutable in Java:**

**StringConstantPool is designed in Java to save memory wherein multiple references can point to same object. Considering this scenario- if one object can be changed(i.e, if string is mutable) then other references who really do not want the value to be changed are also effected, that is why String is immutable in Java.**

**What is the difference between == and equals comparision w.r.to Strings?**

**== - compares the references, == works well with string literals(pool)**

**Equals- compares the actual values**

**String internally represents a char array in memory which starts from 0 index and ends at length-1.**

Keeping efficient memory management in mind- java people created string constant pool mechanism.

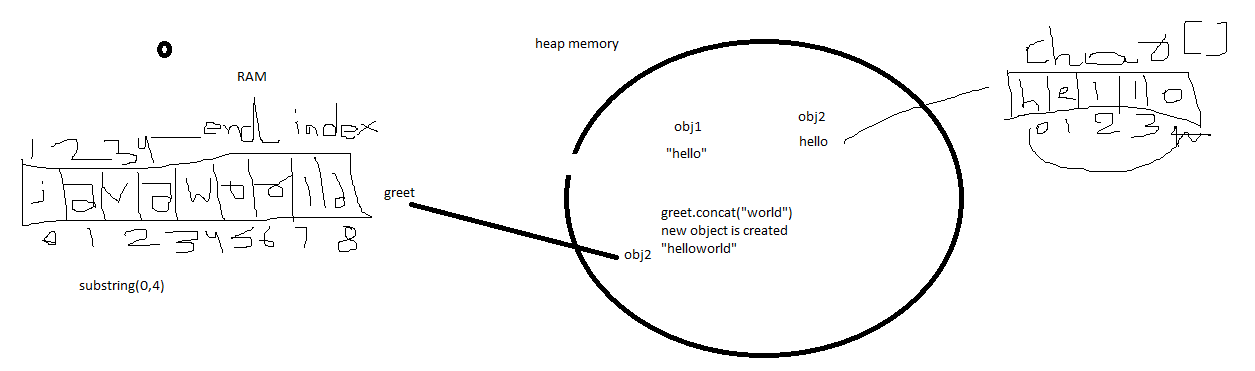
Why string immuatable?

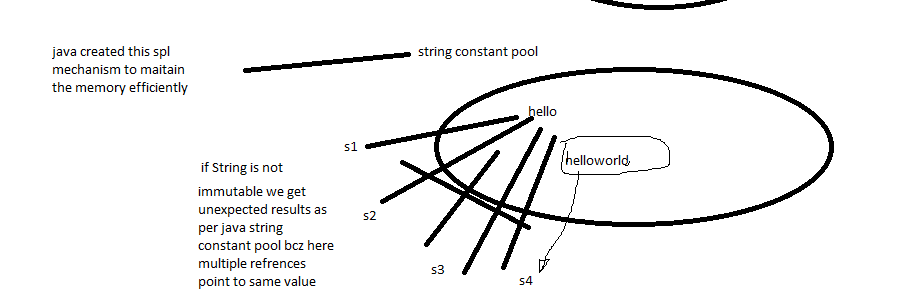
Keeping stringconstant pool in mind they made string immutable.

Other reasons are for security, cache etc…

How u can make any class immutable in java?

By making class final and there are some other rules…and String class is declared final in java to make it immutable





■ charAt() Returns the character located at the specified index

■ concat() Appends one String to the end of another ( "+" also works)

■ equalsIgnoreCase() Determines the equality of two Strings, ignoring case

■ length() Returns the number of characters in a String

■ replace() Replaces occurrences of a character with a new character

■ substring() Returns a part of a String

■ toLowerCase() Returns a String with uppercase characters converted

■ toString() Returns the value of a String

■ toUpperCase() Returns a String with lowercase characters converted

■ trim() Removes whitespace from the ends of a String

StringBuffer and StringBuilder:

These classes are mutable unlike String class.

StringBuffer methods are synchronized , they are thread safe.

StringBuilder methods are not synchronized, they are faster.

StringBuffer st = new StringBuffer(“hello”);

st.append(“world”);

StringBuilder st = new StringBuilder(“hello”);

st.append(“world”);

eg: when dealing with file data appending and whenever we have to modify string frequently we prefer stringbuffer and stringbuilder classes.

append

delete

insert

reverse

**Object is the root class in Java class hierarchy. By default every object created inherits from Object class and it can use Object class methods.**

**Imp Object class methods:**

**equals()**

**hashcode() – unique number for every object**

**toString()**

**wait**

**notify**

**notifyAll**

**Wrapper Classes : To** use objects in place of primitives, and the Java platform provides *wrapper* classes for each of the primitive data types. These classes "wrap" the primitive in an object. Often, the wrapping is done by the compiler—if you use a primitive where an object is expected, the compiler *boxes* the primitive in its wrapper class for you. Similarly, if you use a number object when a primitive is expected, the compiler *unboxes* the object for you

**Datatypes:**

|  |  |  |
| --- | --- | --- |
| **Primitive data types** | **Wrapper Classes-Objects** |  |
| **byte** | **Byte** |  |
| **char** | **Character** |  |
| **int** | **Integer** |  |
| **float** | **Float** |  |
| **long** | **Long** |  |
| **double** | **Double** |  |
| **boolean** | **Boolean** |  |
| **short** | **Short** |  |

There are three reasons that you might use a WrapperClasses rather than a primitive:

1. As an argument of a method that expects an object (often used when manipulating collections of numbers).
2. To use constants defined by the class, such as MIN\_VALUE and MAX\_VALUE, that provide the upper and lower bounds of the data type.
3. To use class methods for converting values to and from other primitive types, for converting to and from strings, and for converting between number systems (decimal, octal, hexadecimal, binary).

**Why WrapperClasses:**

**Gives you object definition for your primitive datatypes due to which you can access them in your Collections or anywhere you need them as objects. Also we can use common methods which come with wrapper classes.**

**It gives you some utility methods to deal with conversions etc.**

**Collections eg:**

**List<Integer> l = new ArrayList<Integer>();**

**List<String> l = new ArrayList<String>();**

**List<Long> l = new ArrayList<Long>();**

**utility methods to deal with conversions etc.**

**Eg: String str = “10”**

**Int I =Integer.parseInt(“10”);**

**Utility methods:**

**valueOf(primitive or string)- creates us Wrapper class object**

**parseXXX(string) - takes string input and returns respective primitive data type.**

**XXX - all wrapper classes**

**xxxValue()- called using our wrapper object and it gives us any primitive value we neeed.**

**Xxx - int, float,long…**

**toString()- converts wrapper obj to string**

**And also we have overloaded toString(primitive type) – converts to string**

**Widening/typecasting – is done with your primitive data types- Byte->short->int->long… but not wrapper classes**

**AutoBoxing:**

**Boxing – is done w.r.to your wrapperclass objects to convert only respective primitive data types to objects**

**int—Integer**

**long—Long**

**Unboxing- converts wrapper to primitive**

**Integer-int**

**Long-long**

**Widening and Boxing: both widening and boxing together are not possible**

**int—Long – NOT possible**

**Thread- is a process in java which runs concurrently.**

**Thread can be created in 2 ways:**

**1.By extending Thread class**

**2.By implementing Runnable interface**

**Thread execution starts when w call start method which invokes run method.**

**synchronized- is key word used to prevent a method being accessed from multiple threads**

**at same time. Which means only one thread can access it and other thread waits until its**

**execution is done.**

**Assignment 7:**

**Write a program to find position of given character in a string- both first occurance and last occurance.**

**Write a program to reverse given string(other than logic discussed in class)**

**Write a program to find lengthiest string in array of strings.**