# Object as a Superclass

public boolean equals(Object obj)  
      Indicates whether some other object is "equal to" this one.

public final Class getClass()  
      Returns the runtime class of an object.

public String toString()  
      Returns a string representation of the object.

**The equals() Method**

* Compares two objects for equality and returns true if they are equal.
* The equals() method provided by Object tests whether the object references are equal – that is, if the objects are compared are the exact same object.

Book firstBook = new Book("0201914670");

Book secondBook = new Book("0201914670");

if (firstBook.equals(secondBook)) {

System.out.println("objects are equal");

} else {

System.out.println("objects are not equal");

}

## The finalize() Method

* finalize() may be invoked on an object when it becomes garbage.
* The finalize() method may be called automatically by the system, but when it is called, or even if it is called, is uncertain.

## The getClass() Method

## The getClass() method returns a Class object, which has methods you can use to get information about the class.

void printClassName(Object obj) {

System.out.println("The object's" + " class is " +

obj.getClass().getSimpleName());

}

## The toString() Method

## The Object’s toString() method returns a String representation of the object, which is very useful for debugging.

## Is override of class method return String type

## System.out.println(s1); similar to s1.toString();

## public void println (Object theObject){ System.out.println(theObject.toString()); }

**Methods**

* A set of code which is referred to by name and can be called (invoked) at any point in a program simply by utilizing the method’s name.
* Collection of statements that are grouped together to perform operations
  + Int number = Console.readInt(“Enter a number”);
  + readInt is the method name

# Constructor

* Special type of method that is used to initialize the object.
* Invoked at the time of object creation.

class Laugher{

public String default;

public Laugher(){

}

public Laugher(String s){

default = s;

}

public void laugher(){

System.out.println(default);

}

}

* Constructor name must be same as class name
  + class Name(){ }
* Called using “new”
  + Class obj = new Class();
* If constructor is invoked again, (using new), the first object is discarded and an entirely new object is created

# Classes

* A template that describes the behavior that objects of its type support.
  + A blue print from which individual objects are created.
  + public class Animal{ }

**Local variables:** variables defined inside methods, constructors or blocks.

public static void main(String[] args){

int keys = 88;

}

**Instance variables:** variables within a class but outside any method.

Class Dog{

public String name; //instance variable

public void doSomething ( String name){

this.name = name; //instance method

}

How to access an Instance Variable and Methods:

ObjectReference = new Constructor();

ObjectReference.variableName;

ObjectReference.MethodName();

**Class variables:** variables declared with in a class, outside any method, with the static keyword.

# Object

* States and behaviors.
* Object’s state is stored in fields and behavior is shown via methods.
* D**eclaration:** A variable declaration with a variable name with an object type.
* **Instantiation:** The 'new' key word is used to create the object.
* **Initialization:** The 'new' keyword is followed by a call to a constructor.

public class Puppy{

public Puppy(String name){

//This constructor has one parameter, name.

System.out.ptrintln(“Pass Name is:” + name\_;

}

public static void main (String []args){

//Following statement would create an object myPuppy

Puppy myPuppy = new Puppy ( “tommy” );

}

}

## Abstract Class

* Class which contains the abstract keyword in the declaration
* Has at least one abstract method
* If a class have at leaste one abstract method, then the class **must** be declared abstract.
  + public abstract class Employee

## Abstract Methods

## A method that is declared without an implementation, just a method signature

## public abstract class Employee{

## private String name;

## private String address;

## private int number;

## public abstract double computePay();

## }

* Any class inheriting the current class must either override the abstract method or declare itself as abstract.

## Concrete Class

## Any such class which has implementation of all of its inherited members ither from interface or abstract class.

## Has no abstract method

## Parameter

* technical term for the value between the round brackets of your method headers.
* Local variable that is set equal to value of its argument
  + void method (int x, int y){

local variable //parameter

}

## Default (packaged) Access Modifier

## Do not explicitly declare an access modifier for a class, field, method.

## Package access can be accessed by name inside the definition of any class in the same package.

## More restricted than protected

## Private Access Modifier (read only) -

* Methods, variables and constructors that are declared private can only be accessed within the declared class itself.
* An instance variable/method cannot be accessed by name outside of class.
* Class and interfaces cannot be private.

public class Logger {

private String format;

public String getFormat() {

return this.format;

}

public void setFormat(String format) {

this.format = format;

}

}

## Public Access Modifier (read and write) +

* Class, method, constructor, interface declared public can be accessed from any other class.
* No restriction on where instance variables/method can be used.

## Protected Access Modifier #

## Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses can be accessed only by the subclasses in other package or any class within the package of the protected members’ class.

## class AudioPlayer {

## protected Boolean openSpeaker( Speaker sp) {

## }

## }

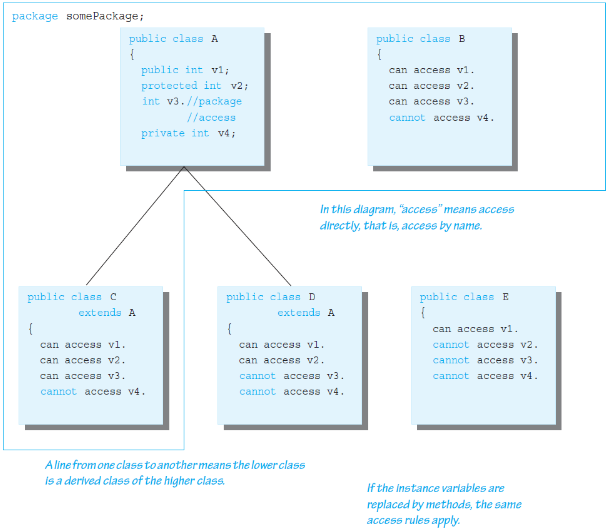
## class StreamingAudioPlayer {

## boolean openSpeaker(Speaker sp){

## }

## }

## If we define *openSpeaker()* method as private, then it would not be accessible from any other class other than *AudioPlayer*. If we define it as public, then it would become accessible to all the outside world. But our intension is to expose this method to its subclass only, thats why we used *protected* modifier.



### Accessor Methods (get)

* Used to return the value of a private field.
* Allows programmer to obtain value of an object’s instance variables

public String getFirstName(){

return firstName;

}

public String getMiddleNames(){

return middleNames;

}

public String getLastName(){

return lastName;

}

public class PersonExample {

public static void main(String[] args){

Person dave = new person(“Dave”, “Bob Bill”, “Davison”, “12 Pall Mall”);

System.out.println(dave.getFirstName() + “ “ + dave.getMiddlesNames() + “ “ + dave.getLastName());

}

}

### Mutator Methods (set)

* used to set a value of a private field.
* allows programmer to change the value of an object’s instance variable in a controlled matter

public void setAddress(String address){

this.address = address;

}

public void setUsername(String username){

this.username = username;

}

public class PersonExample {

public static void main(String[] args){

Person dave = new person(“Dave”, “Bob Bill”, “Davison”, “12 Pall Mall”);

dave.setAddress (“256 Bow Street”);

dave.setUsername(“DDavidson”);

}

}

# this parameter keyword in java

* this is a reference variable that refers to the current object.
* Must be used if a parameter or other local variable with the same name is used in the method.

public class Person{  
 private string fName;

private string lName;

private string uName;

public Person(string fName, string lName){

this.fName =fName;

this.lName=lName;

}

public Person()

{

this(“John”,”Smith”);

}

public void setUsername(string uName){

this.uName = uName;

}

public void setUsername(){

this.setUsername(lName+fName.charAt(0));

}

}

# Composition “has – a”

* using instance variables that are references to other objects
  + class Fruit{

//….

}

class Apple{

private Fruit fruit = new Fruit();

}

* class Apple is related to class Fruit by composition, because Apple has an instance variable that holds a reference to a Fruit object.

# Inheritance “is – a”

* Allows a class to use the properties and methods of another class.
  + class Fruit{

//…

}

class Apple extends Fruit {

//…

}

### static variable

* Only one instance of a static field exist, no matter the amount of instances of the class are created.
  + public static final int BIRTH\_YEAR = 1954;
    - int year = MyClass.BIRTH\_YEAR;
* When referring to defined constant outside class, use class name.

public class Person{

private static int count =0;  
 private string fName;

private string lName;

private string uName;

private int ID;

public Person(string fName, string lName){

this.fName =fName;

this.lName=lName;

ID = count++;

}

public Person()

{

this(“John”,”Smith”);

}

public void setUsername(string uName){

this.uName = uName;

}

public void setUsername(){

this.setUsername(lName+fName.charAt(0));

}

}

public static void main(String[]args){

Person foo = new Person(“John”,”Smith”); //ID will equal 1

Person bar = new Person(“OMG”,”WTFBBQ”); //ID will equal 2

}

### static method

* Methods that use no instance variable of any object of the class they are defined in.

class Languages{

public static void main(String[] args) {

display();

}

static void display(){

System.out.println(“Java is my favorite programming language.”);

// Java is my favorite programming language will be the output.

}

}

* + static methods are invoked using the class name in place of calling object.

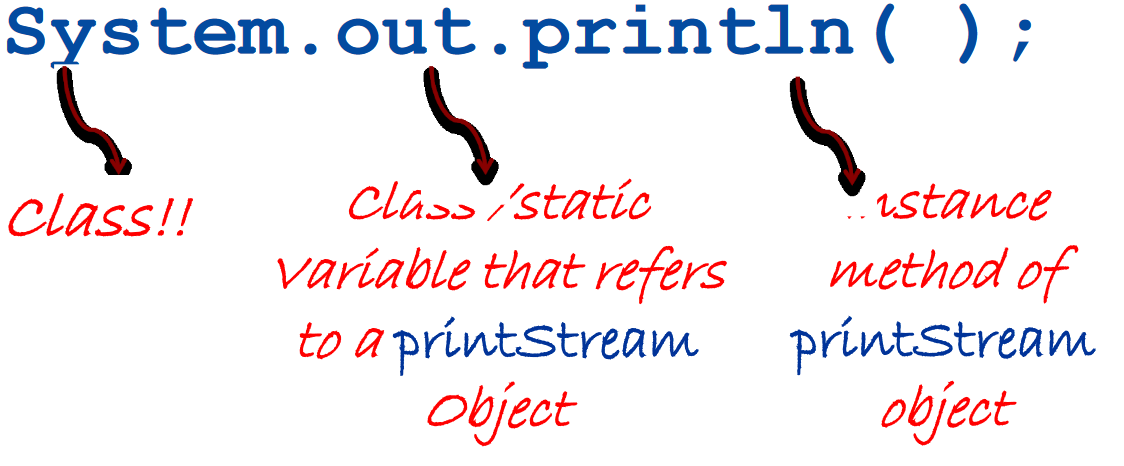
public static returnedTypeMethod(parameters){

}

returnedValue = MyClass.myMethod(argument);

# Java.lang.Math

* double x = Math.random();
* Math.PI //contant is a double
* area of circle = Math.PI \* (radius \* raidus);



# Wrapper class

* wrapper class wraps (encloses) around a data type and gives it an object appearance.

# Boxing

* done by creating an object of the correspoinding wrapper class passing the primitive type value as an argument to the wrapper class constructor.
  + Integer anIntegerObject = new Integer(25);
  + Double aDoubleObject = new Double(3.6);
  + Character aCharacterObject = new Character(‘A’);

# Unboxing

* Invoking specific methods of the wrapper class that return the primitive data type value corresponding to an object of a wrapper class.
  + int I = anIntegerObject.intValue();
  + double d = aDoubleObject.doubleValue();
  + char c = aCharacterObject.charValue();

# Converting String to any primitive data type

* double y = Double.parseDouble("3.17”)

# Converting any primitive data type to String

* String s1 = String.valueOf(‘a’);
  + float myfloatObject = newFloat(3.5);
  + float price = myfloatObject.floartValue();

# Passing Variables

public static void main(String[] args){

String I = “A”;

anotherMetod(i);

}

public static void anotherMethod(int param){

System.out.println(“ “ + param);

}

# ByValue

* make a copy in memory of the actual parameter’s value that is passed in.

# ByReference

* pass a copy of the address of the actual parameter

**== (equal to)**

* Check if the values of two operands are equal or not, if yes then then condition becomes true.

**=**

* Simple assignment operator, assigns values from right side operands to left side operand.

## Copy Constructor

* Constructor that takes only one parameter which is the same exact type as the class in which the copy constructor is defined.

class Employee{

Employree (Employee emp){

employeeName = emp.employeeName;

address = emp.address;

age = emp.age;

salary = emp.salary;

}

### Immutable class

* A class which once created, it’s content cannot be changed.

public final class FinalPersonClass{

private final String name;

private final int age;

public FinalPersonClass(final String name, final int age){

this.name = name;

this.age = age;

}

public int getAge(){

return age;

}

public String getName(){

return name;

}

}

### Mutable class

* A class that contains public mutator methods or other public methods that can change the data in its object.

public class MutableClass{

private int value;

public MutableClass(int aValue){

value = aValue;

}

public void setValue(int aValue){

value = aValue;

}

public getValue(){

return value;

}

}

### Abstraction

* Process of hiding the implementation details and showing only functionality to the user.
  + Avoids information overload

# Encapsulation

* Mechanism of wrapping the data and code acting on the data together as a single unit.
  + Data & methods are inside an object

# Inheritance

* Process where one class acquires the properties of another.

class Super{

}

class Sub extends Super{

}

## super keyword

* Similar to **this**
* Used to definite the members of superclass from that of subclass.

super.variable

super.method();

# Arrays

* Stores a fixed-size sequential collection of elements of the same type.
  + double[] myList;
  + arrayRefVar = new datatype{arraySize};
  + double[] myList = new double[10];

Object[] object Arr = new Object[10];

object Arr[0]=e3;

object Arr[1]=e4;

object Arr[2]=e5;

## Passing Arrays to Methods:

## public static void printArray(int[] array){

## for(int I = 0; i < array/length; i++){

## System.out.print(array[i] + “ “);

## }

## } Returning an Array from a Method:

## public static int[] reverse(int[] list){

## int[] result = new int[list.length];

## for(int i = 0, j = result.length -1; i < list.length; i++, j--){

## result[j] = list[i];

## }

## return result;

## }

## ****While loop****

## Loop that will execute when the Boolean expression is true.

## while( i < 20){

## System.out.ptrint(“value of i : “ + i);

## i++

## }

## }

## ****do..while loop****

## Statement inside loop are executed once before the expression is checked,

## If statement is true, it will go back to the loop and execute statements.

## do{

## System.out,print(“value of u : “ + u);

## u++

## }while( u < 15);

## ****For loop****

## For when task is known to be repeated several times.

## for(int n = 10; n <15; n = n+1){

## System.out.print(“value of n : “ + n );

## }

## ****For each loop****

## Used with iteration over arrays and other such collections.

## int [] numericals = (100, 200, 300. 400, 500)

## for(int u : numericals){

## System.out.print( u);

## }

# Overriding

* A new definition of methods is simply placed in the class definition, just like any other method that is added to the derived class.
* Declaring a method in subclass which is already present in parent class.

## class Human{

## public void eat(){

## System.out.printlb(“Human is eating”);

## }

## }

## class Boy extends Human{

## public void eat(){

## System.out.println(“Boy is eating”);

## }

## public static void main( String args[]){

## Boy obj = new Boy();

## obj.eat();

## }

## }

## Output: Boy is eating

# Overloading

* Allows a class to have two or more methods with the same name, if their argument lists are different.

class DisplayOverloading{

public void disp(char c){

System.out.println( c );

}

public void disp (char c, int num){

System.out.println(c + “ “ + num);

}

}

Class Sample{

public static void main (String args[]) {

DisplayOverloading obj = new DisplayOverloading();

obj.displ( ‘ a ‘ );

obj.disp( ‘a’ , 10);

}

}

**Output:** a

a 10

# [super()](http://stackoverflow.com/questions/3767365/super-in-java)

* Calls the parent constructor with no arguments
  + super(argument1)
  + super.aMethod()

class Vehicle{

Vehicle(){

System.out.println(“Vehicle is created”);

}

}

class Bike5 extends Vehicle{

Bike5(){

super();

System.out.println(“Bike is created”);

}

public static void main(String args[]){

Bike5 b = new Bike5();

}

}

**Output:** Vehicle is created

Bike is created

# Using 'this' keyword in Java constructors

* this(…) is like a method call for constructors
* can use to call another constructor in the same class.

public class Rectangle{

private int x,y;

private int width, height;

public Rectangle(){

this(0, 0, 1, 1);

}

public Rectangle(int width, int height){

this(0, 0, width, height);

}

public Rectangle(int x, int y, int width, int height){

this.x=x;

this.y=y;

this.width = width;

this.height = height;

}

…

}

Use of Private Instance Variables from the Base Class

* An instance variable that is private in a base class is not accessible by name in the definition of a method in any other class, not even in a method definition of a derived class
* Instead, can only be accessed by the public accessor and mutator methods defined in that class

public class Employee{

private String name;

private Date hireDate;

}

public class HourlyEmployee extends Employree{

float wageRate;

int hours;

super.setName(“John”); //used to access name variable

}

# instanceof

* Simple tests the class of an object

public class MainClass{

public static void main(String [] a){

String s = “Hello”;

if (s instanceof java.lang.String){

System.out.printlb(is a String”);}

}

}

**Output:** is a string

# getClass()

* Used with == or != to test if two objects were created with the same class

(object1.getClass()==object2.getClass())

# Java Interfaces

* Like a class, except can only contain method signatures and fields.
* Can be used to achieve polymorphism.
* Before using an interface, it must be implemented in some Java class

public interface MyInterface{

public String hello = “Hello”;

public void say Hello();

}

public class Myinterface implements MyInterface{

public void sayHello(){

System.out.printlb(MyInterface.hello);

}

}

* Interface may be derived from a base interface
  + extend BaseInterfaceName
* Interface derived into class
  + public interface AB extends A, B

# Comparable interface

* Compares two objects, in order to impose an order between them
  + public int compareTo(Object obj);
    - used to compare the current object with specific object.

# ClassCastException

* used to indicate that the application’s code has attempted to cast a specific object to a class of which is not an instance.

try{

}catch (ClassCastException e){

…}

# Serialization

* An object can be represented as a sequence of bytes that includes the object’s data as well as information about the object’s type and the type of data stored in the object.
* Object is serializable if its class or any of its superclasses implements either the java.io.Serializable interface or java.io.Externalizable

public class Employree implements java.io.Serializable{

public String name;

public String address;

public transient int SSN;

public int number;

public void mailCheck(){

System.out.println(“Mailing a check to “ + name + “ “ + address);

}

}

## Serializing an Object

## A way by which an object can be represented as a sequence of bytes that include the object’s data as well as information about the object’s type and the types of data stored in the object.

## Deserializating an Object

## Reads the object data, the type information and bytes that represent the object from the file it was written in and to recreate the object in memory.

# ObjectInputStream

## Provides access to the persistent fields read from the input stream.

## public abstract static class ObjectInputStream.GetField extends Object

## FileInputStream fis = new FileInputStream(“t.tmp);

## ObjectInputStream ois = new ObjectInputStream(fis);

## int i = ois.readInt();

## String today = (String) ois.readObject();

## Date date = (Date) ois.readObject();

## ois.close();

## Serialize an Object and send it to the output stream:

## public final void writeObject(object x)

## throws IOException

# ObjectOutputStream

* Writes primitive data types and graphs of Java objects to an OutpuStream which can be read using ObjectInputStream.
  + public class ObjectOutputStream extends OutputStream implements ObjectOutput, ObjectStreamConstants
  + To write an object that can be read by the example in ObjectInputStream:

FileOutputStream fos = new FileOutputStream(“t.temp”);

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeInt(12345);

oos.writeObject(“Today”);

oos.writeObject(new Date());

oos.close();

* To retrieve the next Object of the stream and deserializes it:

public file Object readObject() throws IOException, ClassNotFoundException

# FileOutputStream

* Output stream for writing data to a File or to a FileDescriptor

public class FileOutputStream extends OutputStream

# FileInputStream

* Reads up to the b.length from file input stream to the array of bytes.

public int read(byte[] b)

* + b – the byte array into which data is read

FileInputStream fis = null;

# Thread Class

* A thread of execution in a program.
  + A thread is a separable computation process

public class Thread extends Object implements Runnable

# Thread.sleep()

* Method that causes the currently executing thread to sleep for the specified number of milliseconds

public static void sleep(long millis) throws InterruptedException

try{

Thread.sleep(1000);

} catch (Exception e){

System.out,println(e);

## RUNNABLE INTERFACE

## a type of interface that can be put into a thread, describing what the thread is supposed to do.

public class MyRunnableTask implements Runnable{

public void run(){

}

}

Thread t = new Thread(new Runnable());

t.start();

# Race Conditions

* Occurs when two or more threads can access shared data and they try to change it at the same time.

# Thread Synchronization

* Making each thread wait so only one thread can run the code in incrememnt() at a time

public synchronized void increment(){

int local;

local counter;

local++

counter = local;

}

## Catching Exceptions

## A try/catch block is placed around the code that might generate an exception

## try{ ….

## }catch(ExceptionName e1){

## //code }

## The throws/throw Keywords:

## If a method does not handle a checked exception, the method must declare it using the throws keyword

## Used to postpone the handling of a checked exception

## public class Name{

## public void withdraw(double amount) throws RemoteExceotion, InsufficientFundsException {

## //method implementation

## }

## //Remainder of class definition

## }

## Exception can be thrown, either a newly instantiated one or an exception that was just caught, by using the throw keyword.

## Used to invoke an exception explicitly

## public class Name{

## public void deposit(double amount) throws RemoteException{

## //method implementation

## throw new RemoteException();

## }

## //Remainder of class definition

## }

# Networking

* **TCP:** Transmission Control Protocol, allows for reliable communication between two applications
  + **TCP** typically used over the internet protocol because It guarantees that data sent from the sender is received in the same order.
  + **Server**: program waiting to receive input
  + **Client**: program that initiates a connection to server

## Sockets

## Describes one end of the connection between two programs over the network.

## Socket Programming

## Most widely used concept in Networking.

## After a connection is established, communication can occur using I/O streams.

# DataOutputStream

## Let’s a primitive to be written to an output source.

## DataOutputSream out = DataOutputStream(OutputStream out);

# accept()

## Listens for a connection to be made to socket and accepts it

## public Socket accept() throws IOException

## Fore server, accept() is typically placed In a loop and a new thread created to handle each client connection while(true){

## Socket connectionSock = serverSock.accept();

## ClientHandler handler = new ClientHandler(connectionSock);

## Thread theThread = new Thread(handler);

## theThread.start();

## }

# close()

## Closes the socket

## public void close() throws IOException

# BufferedReader

## Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, array, and lines

## public class BufferedReader extends Reader

## Buffer the input from specified file

## BufferedReader in = new BufferedReader(new FileRead(“foo.in”));

# readLine()

## Reads a line of text.

## public String readLine() throws IOException

## Will return a String containing the contents of the line.

## Polymorphism

* Allows changes to be made to method definitions in the derived classes and have those changes apply to the code written for the base class.

public interface Vegetarian{}

public class Animal{}

public class Dear extends Animal implements Vegetarian{}

## Binding

## Process of associating a method definition with a method invocation

## Early (static) Binding

## If method definition is associated with its invocation when the code is complied

## Compile time binding

## 

## class Human{

## …

## }

## class Boy extends Human{

## public void walk(){

## System.out.println(“Boy walks”);

## }

## public static void main( String args[]){

## Boy obj1 = new Boy();

## Obj1.walk();

## }

## }

## Late (dynamic) Binding

## Method is associated with its invocation when the method is invoked at run time

## Runtime binding

class Human{

public void walk(){

System.out.printlb(“Human walks”);

}

}

class Boy extends Human{

public void walk(){

System.out.printlb(“Boy walks”);

}

public static void main(String args[])}

Human myobj = new Boy();

Myobj.walk();

}

}

## Upcasting

## When an object of a derived class is assigned to a variable of a base class

## Casting to a supertype

## Downcasting

## When a type cast is performed from a base class to a derived class

## Casting to a subtype