Differences between Java Concepts

Experiment Findings · September 2020

DOI: 10.13140/RG.2.2.30329.49763

1 author:

Dr.Chandrakant Naikodi
Davangere University
88 PUBLICATIONS **50** CITATIONS

SEE PROFILE

Differences between Java Concepts

**** For Computer Science Students of Davangere University ****

Dr. CHANDRAKANT NAIKODI
Associate Professor, DoS in Computer Science
Davangere University, Davangere-577007, India

Chapter 1

Differences between Java Concepts

In this section, listed out some of the key differences among JAVA functions and/or keywords.

1. Difference between: ***extends* and *implements***

Extends(Class)	Implements(Interface)
Related to single class inheritance, *extends* is for classes.	Related to interface, *implements* is for interfaces.
A class cannot implement a class.	interface cannot implements interface.
class extends class	class implements interface and interface can extend interface
Inheritance does one class to inherit the properties of another class.	*implements* builds up to implement the methods declared in the interface.
By using *extends* you can only extend one class in Java at a time.	But by using *implements* you can implement many interfaces at a time.
When you are using a *extends* class, any methods that you write with the same name will override the super class method.	When you are using *implements* a class, the parent class had nothing more than the method declaration (i.e. there was no code in the method). Therefore you are adding your own method code to what was just a method declaration in the interface but not overriding the parent methods instead you just filling up the method.

Similarities between:***extends* and *implements***
a. Both *implements* and *extends* create subtypes.

2. Difference between: ***Imports*, *Implements* and *Inheritance***

Imports	Implements	Inheritance
It is used to import already compiled libraries. e.g: *import java.util.Random;* Imports are generally coded at the beginning or top of your code.	In Java object-oriented program (OOP), a class cannot inherit from multiple classes. Therefore at any instance of time, only one class can be super class. But you can implement more than one interface. However, a class can inherit another class and implement an interface at the same time. An interface is a class but it has only signature of methods.	Inheritance is a feature of OOP. It allows code reuse and expansion.

3. Difference between: ***Abstract class* and *Interface***

Properties	Abstract class	Interface
Definition	The methods of this class must always be redefined in a subclass called overriding and defining the methods in subclass is mandatory	It specifies a set of methods that an instance must handle
Default behavior	Instance methods in abstract class can implement a default behaviour.	It cannot be achieved and all methods are implicitly abstract.
Visibility of members	It can have members of private, public, protected or none, and also with partial implementation.	It has only public members or none (package) and no implementation.
Inheritance	A class can extend only one abstract class or vice versa.	It provides multiple inheritances. A class can implement several interfaces.
Performance	It is faster compared to interfaces.	It is slow, because it requires extra redirection to find corresponding method in the actual classes.

Abstract or non-abstract methods	This class can have non-abstract methods. Further, non-abstract methods need not be overridden in the subclass.	All methods are implicitly abstract.
Variables	Programmer can declare variables as required (default and instance variables and final variables).	All variables in an interface must be *constants*. They are implicitly declared *public static final*.
Constructor	It can define a constructor.	It cannot define a constructor.
Identity	Classes will determine the behavior of an object.	Interfaces will determine the role of an object.
Constants	You can define both instance and static constants. Both static and instance initializer code can access the constants.	Only static final constants can use them without qualification in classes that implement the interface. This is little confusion when you are using them and it is not obvious where they are coming from since the qualification is optional.
Related or unrelated classes implementation or extends	A third party class must be rewritten to extend from the abstract class.	An interface implementation may be added to any existing third party class.
When best to use?	Abstract class works best, if the various implementations of similar kind, and share a common status and behaviour, Abstract classes are useful in a situation where some general methods should be implemented and specialization behaviour should be implemented by child classes.	If all the implementations share the method signatures, then an interface works best. Interfaces are useful in the situations where all properties should be implemented.
Scalability or adding functionality	If a new method is added to an abstract class, it has a default implementation associated. Then all existing code will continue to work without change.	If you add a new method to an interface, you must track down all implementations of that interface and provide them with a concrete implementation of that method.
Static methods	Allowed	static methods are not allowed. Only public or abstract modifier are allowed.

Static initializer blocks	Allowed	Not allowed
User defined exceptions	Not allowed to define.	User defined exceptions can be defined within an interface
Keyword	Abstract class is a class prefix with an *abstract* keyword followed by *class* definition.	Interface which starts with *interface* keyword.
When both are same?	If fully abstract class where all methods declared as abstract and all fields are public static final, then it is similar to interface.	You cannot achieve similar to abstract class.
Code sharing	Possible	Not allowed
Marker	You can use abstract class also as abstract, but it cannot extend any other class.	You can use interface as marker. An interface having no methods is called as a marker interface. Marker is like an indication, which provides intimation to the JAM or Container. For example, we want to store an object in the file then we have to intimate to the compiler that this object should be stored. This intimation can be done by implementing the interface *java.io.Serializable*.
Variable Initialization	Not necessary.	Mandatory.
Defined or derived	In abstract class, methods are derived.	Interfaces are nothing but defined methods.
Relationship	Abstract classes are used only when there is a "is-a" type of relationship between the classes. With abstract classes, you are grabbing away each class's individuality.	Interfaces can be implemented by classes that are not related to one another. With Interfaces, you are merely extending each class's functionality.
Implementations	If a class is declared to be abstract, it need not implement all of the interface methods.	A class that implements an interface must implement all of the interface's methods.

Similarities between:***Abstract class*** **and** ***Interface***
a) Neither abstract class nor interface can be instantiated.
b) Both can have *final* variables.

c) If the client code invokes using interface/abstract only, then you can easily change the concrete implementation behind it using a factory method.

4. Difference between: ***this* and *super()***

this	*super*()
It is used to invoke a constructor of the same class.	It is used to invoke a *super* class constructor
Refers to the current object instance	It is used to refer to the variables and methods of the *super* class from the current object instance

Similarities between:***this* and *super()***
a) Both are used to invoke the constructor. b) Should be first statement in the constructor.

5. Difference between: ***PATH* and *CLASSPATH***

PATH	CLASSPATH
PATH is an environment variable which is used by the OS to find the executable files.	CLASSPATH is an environment variable which is used by the Java compiler to find the path of classes or path of jar files.
PATH is nothing but setting up an environment for OS. The OS will look in this PATH for executables.	CLASSPATH is nothing but setting up the environment for Java where it finds the compiled classes.
PATH refers to the system while CLASSPATH refers to the Developing Environment.	CLASSPATH set path of jars for compiling classes.

6. Difference between: ***Abstract Class* and *Abstract Method***

Abstract Class	Abstract Method
This is a class that is declared as *abstract*. It may or may not include abstract methods.	An *abstract* method is one that is declared without an implementation. (i.e. without braces and followed by a semicolon).
Abstract classes cannot be instantiated but they can be sub classed.	Abstract methods always filled in subclasses.

When an abstract class is subclassed, the subclass usually provides implementations for all the abstract methods in its parent class. However, if it does not, the subclass must also be declared as *abstract*.	Abstract method is overridden by the full method definitions contained in the sub-classes.
An abstract class does not have to contain an abstract method or abstract properties. However, a class that does contain at least one abstract method or abstract properties is considered to be an abstract class. Therefore it should be identified using the keyword *abstract* (otherwise it will not compile). Similarly, an abstract class can contain methods or abstract properties that are not abstract.	Abstract method can be declared in abstract class or non abstract class. Any class with an abstract method is automatically abstract. Thus it should be declared *abstract*
Attempting to instantiate an object of an abstract class results in a compilation error.	An abstract method in Java is something like a pure virtual function in C++ (i.e., a virtual function that is declared = 0).

Similarities between:***Abstract Class*** **and** ***Abstract Method***
a) All methods in an interface are implicitly abstract. So the *abstract* modifier is not used with interface methods.

7. Difference between: ***abstract*** **and** ***final***

property	*abstract*	*final*
Subclassing	Should be subclassed to override the functionality of abstract methods.	Can never be subclassed as final does not permit.
Method alterations	The functionality of methods of abstract class can be altered in the subclass.	Final class methods should be used as it is by other classes.
Overriding concept	All the abstract methods should be overridden for later use.	Final class cannot be inherited.
Inheritance	Can be inherited.	Cannot be inherited.
Instantiation	Cannot be instantiated.	Can be instantiated.
Abstract methods	Can have abstract methods.	Cannot have abstract methods.
Partial implementation	Few methods can be implemented and a few cannot.	All methods must have implementation.

Nature	It is an incomplete class.	It is a complete class.
Adding extra functionality	Extra functionality to the methods can be added in subclass.	No extra functionality can be added and should be used as it is.
Immutable objects	Cannot create immutable objects (also, no objects can be created).	Immutable objects can be created (e.g. String class).

8. Difference between: ***Object*** **and** ***Object reference***

Object	Object Reference
Class instantiation is called instance of a class.	The created object's address is referred by a variable.
It specifies the state and/or behaviour.	The whole object is identified by the reference.
For example: *Bird b = new Bird("pigeon",2);* Here *new Bird("pigeon",2);* is called dynamic object creation. Objects don't have names. It has types, and locations in memory along with fields and methods. The above statement creates a *new Bird* object in memory, initializing it with the data received as arguments to the constructor.	For example: *Bird b = new Bird("pigeon",2);* When object created, reference of the object *Bird* is assigned to the variable *b*. It is a reference or object type variable which can reference a *Bird* object or subclass of *Bird*.
An object may contain references to other objects.	At any moment provides one reference for one object only. A reference object encapsulating the reference to another object is known as referent.
The garbage collector identifies objects that are no longer in use and reclaims the memory or resources allocated.	The referent of a reference object is specified when the reference object is created. If the garbage collector collects a weakly reachable object, all weak references are set to *null*. So that the referent object can no longer be accessed through the weak reference.

Similarities between:***Object*** **and** ***Object Reference***

a) Subclass variable can't reference a superclass object, but only superclass variable can reference a subclass object. Example: All pigeons are birds but all birds are not pigeons.

b) A program can use a reference queue to create a circumstances, where the program is notified when a certain object is reachable only through reference objects. Upon

notification, the program can proceed with clean-up operations on other related objects to make them eligible for garbage collection.

c) An object is in use if it can be accessed or reached by the program in its current state.

9. Difference between: ***Overloading* and *Overriding***

Overloading	Overriding
Another method in the same class having the same method name, but with different arguments.	A method in a subclass with same method name and same arguments as in the super class.
Method signatures are different.	Method signature is same.
It is a relationship between methods available in the same class.	It is a relationship between methods in super class method and subclass.
Separate methods share the same name.	Subclass method replaces the super class method.
Methods with same name co-exists in same class but they must have different method signature.	Signature remains exactly same including return type.
″is a″ relationship	″has a″ relationship
Does not block inheritance from the super class.	Does blocks inheritance from the super class.
Static or early binding polymorphism.	Dynamic or late binding polymorphism.
Simply involves having a method with the same name within the class.	You can change the method behaviour in the derived class.
Overloaded methods are independent methods of a class and can invoke each other just like any other method.	Overriding methods maintains relationship between super class and subclass. It can call the overriding method in super class using keyword *super*.
Constructors can also be overloaded in the similar fashion.	Cannot override constructors.
Overloading is generally used while implementing several methods that implement similar behaviour but for different data types.	The method in the super class is said to be overridden by the method in the subclass. Overriding method actually replaces the super class method with subclass method.
Any visibility is allowed on overloading methods.	Methods overriding cannot be declared more private than the super class method. In other words the overriding method may not limit the access of the method it overrides.

You can declare overloading method as *final*.	Methods declared as *final* cannot be overridden. An overriding method declared as final using keyword *final* suggest that this method cannot be further overridden.
Private method overloading is allowed.	Methods declared as *private* cannot be overridden as they are not visible outside the class.
Method calling is achieved through object	Overriding method can call the overridden method (just like any other method) in its super class with keyword *super*. *super.method*() will invoke the method of immediate super class. Though keyword *super* is used to refer super class, method call *super.super.method*() is invalid.
Overloaded methods need not have the same list of thrown exceptions.	The overriding method should throw same exceptions that are thrown by the overridden method. In other words any exceptions declared in overriding method must be of the same type as those thrown by the super class, or a subclass of that type.
Static overloading methods are allowed.	Static overriding methods are not allowed.

10. Difference between: ***Instance, Object, Reference* and *Class***

Instance	Object	Reference	Class

An instance should have a class definition. Example: *Student S = new Student();* Where *S* is an instance.	An object is an instance of a class. We can create object in different ways as follows • Using *new* operator • Using *class.forName*: *Classname obj = Class.forName("Fully Qualified class Name").newInstance();* • Using *this.getClass().getClassLoader() .loadClass(com.abc.Test).newInstance();* • Using *object.clone* • Using *ObjectInputStream obj =new ObjectInputStream(inputStream); Test obj1 = (Test) obj.readObject();.*	A reference is just like a pointer pointing to an object.	Class is a user defined data type with set of data members and member functions.
This represents the values of data members of a class at a particular instant.	An object is an instance of a class.	*Student S = new Student();* Returns a reference of *Student* object to the *Student* instance.	Class is a template for objects.

11. Difference between: ***""(empty strin)*** **and** ***null***

property	*empty string("")*	*Null*
String	It is a String.	Need not be a String.
.length()	"".length() results in 0.	null.length() results in NullPointerException.
.equals	"".equals(null) returns false because "" does not equal null.	null.equals("") results in NullPointerException.

12. Difference between: ***Literals*** **and** ***Variables***

Literals	Variables

Mentioning truth or actual value.	On demand value generating.
Static value.	Dynamic value.
It could be a constant value.	Values can be varied.
e.g.: *int i = 10;* here 10 is a literal.	e.g.: *int i=k*m*n;* here *k,m* and *n* are variables .

Similarities between:***Literals* and *Variables***
Allows all data types of literals and variables.

13. Difference between: ***Local, Instance* and *Class Variables***

Questions	Local variable (method or block variables)	Instance variable (member variables)	Class variable (Static variables)
How to Define?	These are declared inside a method, constructor, or block. Local variables are not visible outside the method or block. When the method is called, memory is allocated to the local variables. When the method exits, the memory allocated to the local variables is removed and claimed back to memory heap.	These are declared inside the class, but outside the methods. An instance variable is created when an object is created and destroyed when the object is destroyed. When an object is allocated memory from heap, memory is allocated for all class variables	These are variables declared with the *static* keyword in a class, but outside a method. Only one copy per class are created regardless of how many objects are created. They are stored in static memory. When the object is destroyed, the memory allocated to static variables is not removed
What is the lifetime ?	Created when method or constructor is invoked. Destroyed on exit.	Created when the instance of class is created with *new* keyword. Destroyed when there are no more references to enclosing object (made available for garbage collection).	Created when the program starts. Destroyed when the program stops.
Where is it used ?	To do temporary or local computations inside the method or block.	These are essential parts of an object's state. Instance variables hold values that would be accessed by methods.	Widely used for variables whose value or state to be continued throughout the life of the class.

What is scope/visibility ?	Local variables/formal parameters are visible within the method or constructor or block in which they are declared.	Instance variables (fields) can been accessed by all methods in the class.	Similar to instance variable, but are often declared public to make available to users of the class.
Where it could be declared ?	Block, method and constructor	Inside class and outside a method.	Inside class (declared with *static* keyword).
Where to do declaration ?	Anywhere in a method or block, but it has to be declared before it is used.	Anywhere at class level.	Anywhere at class level with *static*.
How to access from outside ?	Not possible to access outside of block-/method.	Instance variables are basically used for information hiding, hence it is usually being declared as private, therefore can not be accessed from outside a class. However to access from outside the class, they must be qualified by an object (eg, Test.t).	These variables are qualified with the class name. They can also be qualified with an object.
What is initial value ?	There is no defined value. Must be assigned a value before the first use.	0(Zero)/false/*null* for numbers, booleans and object respectively. Can be assigned value at declaration or in constructor.	Similar to instance variable and in addition, the value can be assigned in the special static initializer block.

14. Difference between: ***Abstract Method*** **and** ***Static Method***

Abstract method	Static method
Abstract class cannot be instantiated. Abstract methods always implemented in subclasses.	No need to instantiate a class (including abstract by extending etc.) in order to call a static method.

Abstract method can be declared in abstract class or non-abstract class. Any class having an abstract method is abstract class. So class should also be declared abstract. A subclass of an abstract class can be instantiated only if it overrides each of the abstract methods of its superclass and provides method body for all of the subclass. Such a class is generally known as concrete subclass.	Methods or Variables marked *static* belong to the class rather than to any particular instance of the class. These methods or variables can be used without having any instances of that class at all. Only the class variable is sufficient to invoke a static method or access a static variable.
An abstract method can be declared without an implementation (without braces, and followed by a semicolon) and can be defined in the subclass.	Static methods can't be overridden. They can be redefined in a subclass (redefining and overriding are not the same thing). It is known as Hiding.
The static, private and final methods cannot be abstract. Since these types of methods cannot be overridden by a subclass. Similarly, a final class cannot contain any abstract methods.	A static method cannot access non-static/instance variables, because a static method is never associated with any instance.
An abstract class may have static fields and static methods. You can use these static members with a class reference.	Static method can access non-static methods by using instances. By definition, a non-static method is one that is called on instance of some class, whereas a static method belongs to the class itself.

15. Difference between: ***Java compiler*** **and** ***Java interpreter***

Java compiler	Java interpreter
It is a program, which translates Java language source code into the Java Virtual Machine (JVM) bytecodes after compiling a program.	It is a program, which implements the JVM specification and actually executes the bytecodes while running the program.
Command: javac program_name.java	Command: java program_name.
Generates class file while compiling.	Executes class (Java bytecode) file which is generated by Java compiler.
Java compiler reads entire program and checks for errors, then compiles it.	Just-In-Time compiler (or JIT compiler) is a part of a JVM. Its purpose is to take the generic (i.e. cross-platform) bytecodes and compile them into more machine-specific instructions.

16. Difference between: ***Java Compiler*** **and** ***Java Decompiler***

Java compiler	Java decompiler
It is a program, which translates Java language's source code into the Java Virtual Machine (JVM) bytecodes after compiling the program.	It is a program that translates from a low level (generally bytecode) to a higher level source code (human readable class form).

17. Difference between: ***final, finalize*** **and** ***finally***

final	*finalize*	*finally*
Final is a modifier, and this keyword is used for constant declaration.	This keyword is used for garbage collection. *finalize* is used just before object deletion for garbage collection.	This keyword is used in exception handling. The *finally* block always executes soon after control exits from *try* block. This ensures that the finally block is executed even if an unexpected exception occurs while running the program. It allows the developer to avoid having cleanup code accidentally bypassed by either *return*, *continue* or *break*.
The *final* variable acts like a constant. This value can't be changed from its initiated value. The final method can't be overridden. The *final* class cannot be subclassed.	This function can not be used to release non memory resources like file handles, sockets, database connections etc., because Java has only a finite number of these resources and it is not defined when the garbage collection is going to release these non-memory resources through the *finalize()* method.	The function of *finally* will not be executed if you use *System.exit(0)* call. *finally* is a closed exception statement.

A final class implicitly has all the methods as final, but not necessarily the data members. A *final* class may not be extended or final method be overridden.	The finally block code is guaranteed of execution irrespective of occurrence of exception. Whereas for finalize it is not guaranteed. The *finalize* method is called by the garbage collector on an object when the garbage collector determines that there are no more references to the object.	Basically finally block is used to release resources irrespective of exceptions. It will be executed whether or not *try* block executes.

18. Difference between: ***Constructor* and *Default Constructor***

Constructor	Default Constructor
Similar to methods which has same name as class. Constructor does not have return statement or return type. When a new instance (a new object) of a class is created using *new* keyword, a constructor for that class is invoked. Generally used to initialize the instance variables (fields) of the object.	It is a constructor that is automatically generated in the absence of explicit constructors or defined constructor. If you don't define a constructor for a class, the compiler automatically creates a default parameterless constructor. The default constructor calls the default parent constructor (*super()*) and initializes all instance variables to default value (zero for numeric types, null for object references, and false for booleans). Default constructor is created only if there are no constructors. If a constructor is defined for the class, then default constructor is not created.

The first line of a constructor must either be a call on to another constructor in the same class (using this operator), or a call to the *super* class constructor (using *super* method). If the first line is neither of these, the compiler automatically inserts a call to the parameterless *super* class constructor. *this* operator can be used invoke another constructor in same class. *super()* can be used to call a constructor of a parent class.	This default constructor will call the no argument constructor of the super class. In this case, the compiler will throw error if the super class doesn't have a no argument constructor. So it should be verified for the existence before invoking no argument constructor. If your class has no explicit super class, then it has an implicit super class of *Object*, which does have a no-argument constructor. A default constructor is called as *nullary* constructor and does not take any arguments.

19. Difference between: ***Class* and *Interface***

Class	Interface
A class is a group of methods and variables.	An interface provides signatures of the methods. Unlike classes, interfaces do not provide their definitions. Classes define all methods.
The class automatically extends the class object.	By default all methods are abstract. Hence, do not use *abstract* keyword before the *abstract* method.

20. Difference between: ***Compile time errors* and *Run time errors***

Compile time errors	Run time errors
Mostly determines syntax errors.	An error, which happens while running a program.
Error may be due to the structure of Java language.	Runtime errors are due to logical errors.
Example: Missing data type for a variable.	Example: A number divides zero.
Common mistakes: Syntax errors or semantic errors. Type checking errors.	Common mistakes: Casting a null pointer. Running out of memory. Trying to access a file that isn't there.

21. Difference between: ***Errors*** **and** ***Exceptions***

Errors	Exceptions
Generally, error refers to compile time error. Example: Can't convert *xxx* to *Boolean*	Runtime error is called an exception. Example: An *ArithmeticException* is a *RuntimeException*. An *ArithmeticException* is a recoverable error.
Errors that can't be handled by the JVM will to lead to termination of the program. It needs to be corrected in source code.	These errors are in the scope of JVM and can be handled using try-catch block. A checked exception is something that might be captured at runtime and can be handled appropriately.
The error class defines error conditions that should not attempted to recover. It is advisable to terminate when such an error is encountered.	The exception class defines mild error conditions that the program encounters. Exceptions can occur when trying to open the file, which does not exist or operands being manipulated are out of prescribed ranges or the class file you are interested in loading is missing, etc..

Similarities between:***Errors*** **and** ***Exceptions***
a). *Exception* and *Error* is sub class of *Throwable*.

22. Difference between: ***try/catch/throw*** **and** ***try/catch(e)/throw e***

try/catch/throw	try/catch(e)/throw e

Syntax:
`try { }`
`catch {`
`throw;`
`}`
	Syntax:
	`try { }`
	`catch(Exception e) {`
	`throw e;`
	`}`

The current exception is thrown again and that exception will keep its "source" and stack trace.	When exception is thrown the source and stack trace is changed. An exception can be thrown from the current method.

Similarities between:***try/catch/throw*** **and** ***try/catch(e)/throw e***
It will catch every exception thrown inside the *try* block.

23. Difference between: ***Instance method*** **and** ***Class method***

Instance method	Class method
An instance method is a regular method of an instantiated class. In other words, the methods of the instantiated object of a class.	A class method is a static method. It is called using the class name.
A class method is associated with a particular object instance. It has 'this' reference.	A class method is not associated with a particular object instance. It does not have a 'this' reference.
Instance of a class is created using *new* keyword.	Declared with the keyword *static*.

24. Difference between: ***Mutable Objects*** **and** ***Immutable Objects***

Mutable Objects	Immutable Objects
When you have a reference to an instance of an object, the contents of that instance can be altered. e.g., *stringbuffer obj*	When you have a reference to an instance of an object, the contents of that instance cannot be altered. e.g,.*String obj*

25. Difference between: ***Throw, Throws*** **and** ***Throwable***

Throw	Throws	Throwable
The keyword *throw* is used to throw user defined exceptions.	The keyword *throws* is used for method signatures to declare that the current method would possibly throw an exception.	Throwable is an interface that the *Exception* class implements, and all user defined class would implicitly implement an interface to ensure that they have exception like behavior.

A *throw* statement requires single argument. That is a instance of any subclass of the *Throwable* class. Executing *throw* statement triggers the JVM to throw this exception and causes an exception to occur.	For checked exceptions, the compiler will guarantee the code invoking that method must catch these checked exceptions.	All errors and exceptions classes are derived from *java.lang.Throwable* class. Only objects that are instances of this class (or one of its subclasses) are thrown by JVM or can be thrown by *throw* statement.

26. Difference between: ***final class*** **and** ***abstract class***

final class	*abstract* class
Final class is a class which can't be subclassed.	Abstract class should be sub-classed. Abstract class can contain both abstract methods and non-abstract methods.
Final class must be used without any modification.	Sub-class of the abstract class is used.
We can create object.	We cannot create object and cannot be instantiated.
final Class cannot be overridden.	Abstract methods must be overridden when used by subclass. It usually defines some default implementations and provides some tools useful for a full implementation.
Final class cannot be extended and cannot be inherited.	Abstract class can be extended.

27. Difference between: ***Call by value*** **and** ***Call by reference***

Call by value	Call by reference
Java supports only call by value.	Java doesn't pass methods arguments by reference.
Primitive types passed in the method call to push their values on stack and hence called as "called by values".	All objects passed to any method call, pass their reference on stack and hence called as "called by reference".
Any changes it makes to those values have no effect on the caller's variables.	The object reference itself is passed by value and so both the original reference and parameter copy refer to the same object.

28. Difference between: ***break statement*** **and** ***return statement***

break	return
break exits out of the current loop.	*return* exits from the function.
Used in loops and *switch* statement.	Used inside the method.

29. Difference between: ***break statement*** **and** ***continue statement***

break	continue
break exits out of current loop or block.	Flow of control skips the following statements inside current loop.
Used in loops and *switch* statement.	Used only in loops.

30. Difference between: ***Procedural programming*** **and** ***Object-oriented programming***

Procedural	Object-oriented
Programming logic follows certain procedures and the instructions are executed one after another.	Unit of program is object, which is combination of data and methods to handle data.
Data is exposed to the whole program.	Data is accessible within the object and it assures data security.

31. Difference between: ***Super class*** **and** ***Subclass***

Super class	Subclass
A super class is a class that will be inherited.	Subclass is a class that does the inheriting.
super class is a base class.	A subclass can inherit the properties of a super class.
super class can be extended by subclasses.	A subclass can access all the methods and variables of super class. A subclass extends a super class.

32. Difference between: ***Access specifier* and *Access modifier***

Access specifier	Access modifier
They are used to set the visibility of a class or variable or a method. Classes can be *public* or *default*. Variables and methods can be *public, private, protected* or *default*. Access ranges from totally accessible to totally inaccessible.	Essential to declare field. You can optionally declare a field with following modifier keywords: *final* or *volatile* and/or *static* and/or *transient*. They are optionally used to declare a field. Access modifiers can be applied for class methods or variables. Classes can have modifier like *abstract*. Methods can be abstract, native, synchronized etc.
The access specifiers of Java are *public, private, protected* or default.	Access modifiers provided by Java are *public, private, protected* or default.

33. Difference between: **== and *equals()***

==	equals()
Used to compare two numbers.	Used to compare two strings.
It can be used to compare the references of the objects.	This method can be overridden for *String* class. The *equals*() method can be used to compare the values of two objects.
This is used to check if two variables point at the same instance of a *String* object.	This is used to compare the values of the *Strings*.
No overriding applies.	On overriding *equals*(), the *hashCode()* method should also be overridden.

34. Difference between: **& and &&**

&	&&
”bit-wise AND” operator	”Conditional logical AND” operator.
Ampercent (&) can be used as Bitwise operator.	AND operation is used for comparison operations.
Evaluates both arguments.	Evaluate the second argument, if the first argument is true.

35. Difference between: | **and** ||

|	||
"bit-wise OR" operator	"Conditional logical OR" operator.
Evaluates both arguments.	Evaluate the second argument if the first argument is false.

36. Difference between: ***JRE* and *JDK***

JRE	JDK
Used for running environment	Used for development environment
It is an implementation of the JVM which actually executes Java programs.	It is a bundle of software that you can use to develop Java based applications.
Java Run Time Environment (JRE) is a plug-in needed for running Java programs.	Java Development Kit (JDK) is needed for developing Java applications.
JRE is smaller than JDK. Hence it required less disk space compare to JDK.	JDK needs more disk space as it contains JRE along with various development tools.
It includes JVM, core libraries and other additional components to run applications and applets written in Java.	It includes JRE, API classes, Java compiler, webpages and additional files needed to write Java applets and applications.

37. Difference between: ***Labelled break statement* and *Unlabelled break statement***

Labelled *break* statement	Unlabelled *break* statement
It terminates and transferes the control to the statement following label.	It terminates the innermost *switch*, *for*, *while*, or *do-while* loop / block.

Example:	Example:

Label:
for(i=0;i<arr.length;i++){
for(j=0;j<arr[i].length;j++){
if (arr[i][j] == searchfor){
break Label;
}
}
}
	for(i=0;i<arr.length;i++) {
	for(j=0;j<arr[i].length;j++){
	if (arr[i][j] == searchfor){
	break;
	}
	}
	}

38. Difference between: ***length* and *length()***

length	length()
The *length* field is used to find out the array length/size.	The *length*() method is used to find out the *String* length.
Example: String s[]={new String("AB"), new String("CD")}; int j=s.length; System.out.println("j="+j);	Example: String str=new String("AB"); int i=str.length(); System.out.println("i="+i);

39. Difference between: ***JAVA* and *J2EE***

JAVA	J2EE
It is a programming language.	It is a platform and it is a implementation of the Java programming language.
Generally, it is used for business application, internet programming, games, etc.	Generally, it is used for business organizations, enterprise applications and distributed applications, web applications etc.

40. Difference between: ***Tomcat server* and *Web logic server***

Tomcat server	Web logic server
It is a web-server.	It is an application-server.
It is a 3-tier architecture.	It is N-tier Architecture.
It only runs servlets, JSP etc. but we cannot deploy an EJB.	We can deploy an EJB along with servlets and JSPs.
Handles only HTTP protocol.	Handles FTP and HTTP.

41. Difference between: ***JRE* and *JAR***

JRE	JAR
Java Runtime Environment.	Java ARchive.
It includes Java virtual machine and some other library files.	It contains the classes, images, sound files, help files, etc for a Java application.
It does not contain any development tools such as compiler, debugger, etc.	It is in ZIP file format.
It runs a Java application.	It is used to distribute Java applications or libraries, in the form of classes, and associated metadata and resources.