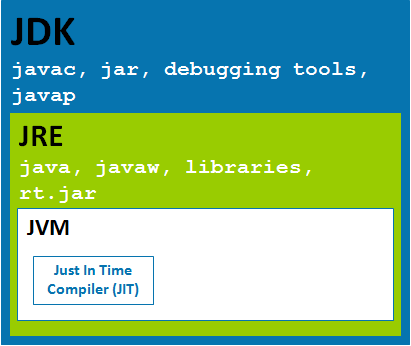
**Java Reviewer**

1. What is the difference between JDK, JRE and JVM?

[](http://cdn.javabeat.net/wp-content/uploads/2013/02/jvm-jre-jdk.png)

JDK (Java Development Kit)

Java Developer Kit contains tools needed to develop the Java programs, and JRE to run the programs. The tools include compiler (javac.exe), Java application launcher (java.exe), Appletviewer, etc… Compiler converts java code into byte code. Java application launcher opens a JRE, loads the class, and invokes its main method.

You need JDK, if at all you want to write your own programs, and to compile the m. For running java programs, JRE is sufficient. JRE is targeted for execution of Java files i.e. JRE = JVM + Java Packages Classes (like util, math, lang, awt, swing etc.)+runtime libraries. JDK is mainly targeted for java development. I.e. you can create a Java file (with the help of Java packages), compile a Java file and run a java file.

JRE (Java Runtime Environment)

Java Runtime Environment contains JVM, class libraries, and other supporting files. It does not contain any development tools such as compiler, debugger, etc. Actually JVM runs the program, and it uses the class libraries, and other supporting files provided in JRE. If you want to run any java program, you need to have JRE installed in the system

The Java Virtual Machine provides a platform-independent way of executing code; programmers can concentrate on writing software, without having to be concerned with how or where it will run. But, note that JVM itself not a platform independent. It only helps Java to be executed on the platform-independent way. When JVM has to interpret the byte codes to machine language, then it has to use some native or operating system specific language to interact with the system. One has to be very clear on platform independent concept. Even there are many JVMs written on Java, however they too have little bit of code specific to the operating systems.

If u just want to run applets (ex: Online Yahoo games or puzzles), JRE needs to be installed on the machine.

JVM (Java Virtual Machine)

As we all aware when we compile a Java file, output is not an ‘exe’ but it’s a ‘.class’ file. ‘.class’ file consists of Java byte codes which are understandable by JVM. Java Virtual Machine interprets the byte code into the machine code depending upon the underlying operating system and hardware combination. It is responsible for all the things like garbage collection, array bounds checking, etc… JVM is platform dependent.

The JVM is called “virtual” because it provides a machine interface that does not depend on the underlying operating system and machine hardware architecture. This independence from hardware and operating system is a cornerstone of the write-once run-anywhere value of Java programs.

There are different JVM implementations are there. These may differ in things like performance, reliability, speed, etc. These implementations will differ in those areas where Java specification doesn’t mention how to implement the features, like how the garbage collection process works is JVM dependent, Java spec doesn’t define any specific way to do this.

1. Object Oriented Programming

Object-oriented programming (OOP) is a programming paradigm that represents the concept of “objects” that have data fields (attributes that describe the object) and associated procedures known as methods.

Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

Object Oriented Programming is a programming paradigm that uses "objects" to design applications and computer programs.

1. What is an “Object”?

An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tangible and intangible). The example of intangible object is banking system.

An object has three characteristics:

* **State**: represents data (value) of an object.
* **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
* **Identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.

|  |
| --- |
| For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior. |

|  |
| --- |
| Object is an instance of a class. Class is a template or blueprint from which objects are created. So object is the instance (result) of a class. |

1. What is a “Class”?

|  |
| --- |
| A class is a group of objects that has common properties. It is a template or blueprint from which objects are created. |

A class in java can contain:

* Data Member
* Method
* Constructor
* Block
* Class and Interface

1. Difference between “Object” and “Class”

| No. | Object | Class |
| --- | --- | --- |
| 1 | Object is an **instance** of a class. | Class is a **blueprint or template** from which objects are created. |
| 2 | Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc. | Class is a **group of similar objects**. |
| 3 | Object is a **physical** entity. | Class is a **logical** entity. |
| 4 | Object is created through **new keyword** mainly e.g. Student s1=new Student(); | Class is declared using **class keyword** e.g. class Student{} |
| 5 | Object is created **many times** as per requirement. | Class is declared **once**. |
| 6 | Object **allocates memory when it is created**. | Class **doesn't allocated memory when it is created**. |
| 7 | There are **many ways to create object** in java such as new keyword, newInstance () method, clone () method, factory method and deserialization. | There is only **one way to define class** in java using class keyword. |

1. Encapsulation

Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods. If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as data hiding.

Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class. Access to the data and code is tightly controlled by an interface.

The main benefit of encapsulation is the ability to modify our implemented code without breaking the code of others who use our code. With this feature Encapsulation gives maintainability, flexibility and extensibility to our code.

Benefits of Encapsulation:

The fields of a class can be made read-only or write-only.

A class can have total control over what is stored in its fields.

The users of a class do not know how the class stores its data. A class can change the data type of a field, and users of the class do not need to change any of their code.

1. Abstraction

Abstraction is process of hiding the implementation details and showing only the functionality.

Use abstraction if you know something needs to be in class but implementation of that varies.

In Java you cannot create instance of abstract class, its compiler error.

Abstract is a keyword in java.

A class automatically becomes abstract class when any of its method declared as abstract.

Abstract method doesn't have method body.

Variable cannot be made abstract, its only behavior or methods which would be abstract.

If a class extends an abstract class or interface it has to provide implementation to all its abstract method to be a concrete class. Alternatively this class can also be abstract.

Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that is declared, but contains no implementation. Abstract classes may not be instantiated, and require subclasses to provide implementations for the abstract methods. Let's look at an example of an abstract class, and an abstract method.

Animals are capable of doing different things like flying, digging and walking, but there are some common operations as well like eating and sleeping. Some common operations are performed by all animals, but in a different way as well. When an operation is performed in a different way, it is a good candidate for an abstract method (forcing subclasses to provide a custom implementation)

Now you may be wondering why not declare an abstract class as an interface, and have the Dog and Cow implement the interface. Sure you could - but you'd also need to implement the eat and sleep methods. By using abstract classes, you can inherit the implementation of other (non-abstract) methods. You can't do that with interfaces - an interface cannot provide any method implementations.

1. Interface

An interface is a contract (or a protocol, or a common understanding) of what the classes can do.

When a class implements a certain interface, it promises to provide implementation to all the abstract methods declared in the interface.

Interface defines a set of common behaviors.

The classes implement the interface agree to these behaviors and provide their own implementation to the behaviors. This allows you to program at the interface, instead of the actual implementation. One of the main usage of interface is provide a communication contract between two objects. If you know a class implements an interface, then you know that class contains concrete implementations of the methods declared in that interface, and you are guaranteed to be able to invoke these methods safely. In other words, two objects can communicate based on the contract defined in the interface, instead of their specific implementation.

Key points:

Here are the key points to remember about interfaces:

1. We can’t instantiate an interface in java.
2. Interface provides complete abstraction as none of its methods can have body. On the other hand, abstract class provides partial abstraction as it can have abstract and concrete (methods with body) methods both.
3. “implements” keyword is used by classes to implement an interface.
4. While providing implementation in class of any method of an interface, it needs to be mentioned as public.
5. Class implementing any interface must implement all the methods, otherwise the class should be declared as “abstract”.
6. Interface cannot be declared as private, protected or transient.
7. All the interface methods are by default abstract and public.
8. Variables declared in interface are public, static and final by default.

interface Try

{

int a=10;

public int a=10;

public static final int a=10;

final int a=10;

static int a=0;

}

All of the above statements are identical.

1. Interface variables must be initialized at the time of declaration otherwise compiler will through an error.

interface Try

{

int x;//Compile-time error

}

Above code will throw a compile time error as the value of the variable x is not initialized at the time of declaration.

1. Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final. Here we are implementing the interface “Try” which has a variable x. When we tried to set the value for variable x we got compilation error as the variable x is public static final by default and final variables cannot be re-initialized.

Class Sample implements Try

{

public static void main(String arg[])

{

x=20; //compile time error

}

}

1. Any interface can extend any other interface but cannot implement it. Class implements interface and interface extends interface.
2. A class can implements any number of interfaces.
3. If there are having two or more same methods in two interfaces and a class implements both interfaces, implementation of one method is enough.

interface A

{

public void aaa();

}

interface B

{

public void aaa();

}

class Central implements A,B

{

public void aaa()

{

//Any Code here

}

public static void main(String arg[])

{

//Statements

}

}

1. Methods with same signature but different return type can’t be implemented at a time for two or more interfaces.

interface A

{

public void aaa();

}

interface B

{

public int aaa();

}

class Central implements A,B

{

public void aaa() // error

{

}

public int aaa() // error

{

}

public static void main(String arg[])

{

}

}

1. Variable names conflicts can be resolved by interface name e.g:

interface A

{

int x=10;

}

interface B

{

int x=100;

}

class Hello implement A,B

{

public static void Main(String arg[])

{

System.out.println(x); // reference to x is ambiguous both variables are x

System.out.println(A.x);

System.out.println(B.x);

}

}

Benefits of having interfaces:

Following are the advantages of interfaces:

Without bothering about the implementation part, we can achieve the security of implementation

In java, multiple inheritance is not allowed, however by using interfaces you can achieve the same. A class can extend only one class but can implement any number of interfaces. It saves you from Deadly Diamond of Death (DDD) problem.

1. Polymorphism

By name we can come to a conclusion that it’s about something with many forms. In OOP it is an ability of an object to take many forms

1. Inheritance

Concept wherein the state and behavior of one class is inherited by another class

1. Difference of Abstract and Interface

|  |  |  |
| --- | --- | --- |
| No. | Abstract | Interface |
| 1 | Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. |
| 2 | Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3 | Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4 | Abstract class **can have static methods, main method and constructor**. | Interface **can't have static methods, main method or constructor**. |
| 5 | Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 6 | The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 7 | **Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

1. Difference of Method Overloading and Method Overriding

|  |  |  |
| --- | --- | --- |
| No. | Method Overloading | Method Overriding |
| 1 | Method overloading is used *to increase the readability* of the program. | Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class. |
| 2 | Method overloading is performed *within class*. | Method overriding occurs *in two classes* that have IS-A (inheritance) relationship. |
| 3 | In case of method overloading, *parameter must be different*. | In case of method overriding, *parameter must be same*. |
| 4 | Method overloading is the example of *compile time polymorphism*. | Method overriding is the example of *run time polymorphism*. |
| 5 | In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter. | *Return type must be same or covariant* in method overriding. |

1. Give me two Types of Modifiers?

Modifiers are keywords that you add to those definitions to change their meanings.

The Java language has a wide variety of modifiers, including the following:

* Java Access Modifiers
* Non Access Modifiers

To use a modifier, you include its keyword in the definition of a class, method, or variable. The modifier precedes the rest of the statement, as in the following examples (Italic ones):

public class className {

private boolean myFlag;

static final double weeks = 9.5;

protected static final int BOXWIDTH = 42;

public static void main(String[] arguments) {

// body of method

}

}

**Access Control Modifiers:**

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

* Visible to the package, the *default*. No modifiers are needed.
* Visible to the class only (*private*).
* Visible to the world (*public*).
* Visible to the package and all subclasses (*protected*).

**Non Access Modifiers:**

Java provides a number of non-access modifiers to achieve many other functionality.

* The *static* modifier for creating class methods and variables
* The *final* modifier for finalizing the implementations of classes, methods, and variables.
* The *abstract* modifier for creating abstract classes and methods.
* The *synchronized* and *volatile* modifiers, which are used for threads.

1. What is a “static” keyword?

The static keyword in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

* **Java static variable**

If you declare any variable as static, it is known static variable.

* The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees, college name of students etc.
* The static variable gets memory only once in class area at the time of class loading.

**Advantage of static variable**

* It makes your program memory efficient (i.e. it saves memory).
* **Java static method**

If you apply static keyword with any method, it is known as static method.

* A static method belongs to the class rather than object of a class.
* A static method can be invoked without the need for creating an instance of a class.
* A static method can access static data member and can change the value of it.

**Restrictions for static method**

* There are two main restrictions for the static method. They are:
* The static method cannot use non static data member or call non-static method directly.
* This and super cannot be used in static context.
* **Java static block**
* Is used to initialize the static data member.
* It is executed before main method at the time of class loading.

1. What is “final” keyword? Give me ways to final keyword

In Java, items with the final modifier cannot be changed!

This includes final classes, final variables, and final methods:

* A *final class* cannot be extended by any other class
* A *final variable* cannot be reassigned to another value
* A *final method* cannot be overridden

A final class is simply a class that can't be extended. *(This does not mean that all references to objects of the class would act as if they were declared as final.)*

When it's useful to declare a class as final is covered in the answers of this question:

Good reasons to prohibit inheritance in Java? If Java is object oriented, and you declare a class final, doesn't it stop the idea of class having the characteristics of objects?

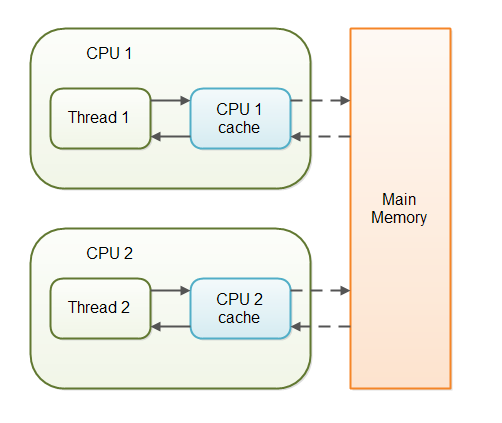
In some sense yes.

By marking a class as final you disable a powerful and flexible feature of the language for that part of the code. Some classes however, should not (and in certain cases cannot) be designed to take sub classing into account in a good way. In these cases it makes sense to mark the class as final, even though it limits OOP.

*(Remember however that a final class can still extend another non-final class.)*

1. What is “volatile” keyword?

The Java volatile keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.



With non-volatile variables there are no guarantees about when the Java Virtual Machine (JVM) reads data from main memory into CPU caches, or writes data from CPU caches to main memory. Let me explain what problems that can cause with an example:

Imagine a situation in which two or more threads have access to a shared object which contains a counter variable declared like this:

public class SharedObject {

public int counter = 0;

}

Thread 1 could read a shared counter variable with the value 0 into its CPU cache, increment it to 1 and not write the changed value back into main memory. Thread 2 could then read the same counter variable from main memory where the value of the variable is still 0, into its own CPU cache. Thread 2 could then also increment the counter to 1, and also not write it back to main memory. Thread 1 and Thread 2 are now practically out of sync. The real value of the shared counter variable should have been 2, but each of the threads has the value 1 for the variable in their CPU caches, and in main memory the value is still 0. It is a mess! Even if the threads eventually write their value for the shared counter variable back to main memory, the value will be wrong.

By declaring the shared counter variable volatile the JVM guarantees that every read of the variable will always be read from main memory, and that all writes to the variable will always be written back to main memory. Here is how the volatile declaration looks:

public class SharedObject {

public volatile int counter = 0;

}

In some cases simply declaring a variable volatile may be enough to assure that multiple threads accessing the variable see the latest written value. I will get back to which cases volatile is sufficient later.

In the situation with the two threads reading and writing the same variable, simply declaring the variable volatile is not enough. Thread 1 may read the counter value 0 into a CPU register in CPU 1. At the same time (or right after) Thread 2 may read the counter value 0 into a CPU register in CPU 2. Both threads have read the value directly from main memory. Now both variables increase the value and writes the value back to main memory. They both increment their register version of counter to 1, and both write the value 1 back to main memory. The value should have been 2 after two increments.

The problem with multiple threads that do not see the latest value of a variable because that value has not yet been written back to main memory by another thread, is called a "visibility" problem. The updates of one thread are not visible to other threads.

**The Java volatile Guarantee**

Since Java 5 the volatile keyword guarantees more than just the reading and writing of a variable from and to main memory. Actually, the volatile keyword guarantees this:

If Thread A writes to a volatile variable and Thread B subsequently reads the same volatile variable, then all variables visible to Thread A before writing the volatile variable, will also be visible to Thread B.

The reading and writing instructions of volatile variables cannot be reordered by the JVM (the JVM may reorder instructions for performance reasons as long as the JVM detects no change in program behavior from the reordering). Instructions before and after can be reordered, but the volatile read or write cannot be mixed with these instructions. Whatever instructions follow a read or write of a volatile variable are guaranteed to happen after the read or write.

Look at this example:

Thread A:

sharedObject.nonVolatile = 123;

sharedObject.counter = sharedObject.counter + 1;

Thread B:

int counter = sharedObject.counter;

int nonVolatile = sharedObject.nonVolatile;

Since Thread A writes the non-volatile variable sharedObject.nonVolatile before writing to the volatilesharedObject.counter, then both sharedObject.nonVolatile and sharedObject.counter are written to main memory.

Since Thread B starts by reading the volatile sharedObject.counter, then both thesharedObject.counter and sharedObject.nonVolatile are read in from main memory.

The reading and writing of the non-volatile variable cannot be reordered to happen before or after the reading and writing of the volatile variable.

**When is volatile enough?**

As I have mentioned earlier, if two threads are both reading and writing to a shared variable, then using the volatile keyword for that is not enough. You need to use [synchronization](http://tutorials.jenkov.com/java-concurrency/synchronized.html) in that case to guarantee that the reading and writing of the variable is atomic.

But in case one thread reads and writes the value of a volatile variable, and other threads only read the variable, then the reading threads are guaranteed to see the latest value written to the volatile variable. Without making the variable volatile, this would not be guaranteed.

**Performance Considerations of volatile**

Reading and writing of volatile variables causes the variable to be read or written to main memory. Reading from and writing to main memory is more expensive than accessing the CPU cache. Accessing volatile variables also prevent instruction reordering which is a normal performance enhancement technique. Thus, you should only use volatile variables when you really need to enforce visibility of variables.

1. What is “Serialization”?

**Serialization** is the conversion of an object to a series of bytes, so that the object can be easily saved to persistent storage or streamed across a communication link. The byte stream can then be deserialized - converted into a replica of the original object

Java provides a mechanism, called object serialization where 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 types of data stored in the object

1. What is “transient” keyword?

Transient is a Java keyword which marks a member variable not to be serialized when it is persisted to streams of bytes. When an object is transferred through the network, the object needs to be 'serialized'. Serialization converts the object state to serial bytes.

1. What is “stream bytes” or “serial bytes”?

Programs use *byte streams* to perform input and output of 8-bit bytes. All byte stream classes are descended from [InputStream](https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html" \t "_blank) and [OutputStream](https://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html" \t "_blank).

There are many byte stream classes. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, [FileInputStream](https://docs.oracle.com/javase/8/docs/api/java/io/FileInputStream.html" \t "_blank) and [FileOutputStream](https://docs.oracle.com/javase/8/docs/api/java/io/FileOutputStream.html" \t "_blank). Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

Java Byte streams are used to perform input and output of 8-bit bytes, whereas Java **Character streams** are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are ,FileReader and FileWriter.. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write above example which makes use of these two classes to copy an input file (having unicode characters) into an output file:

1. What is an “Object Class”?
2. What is the use of “hashCode” in java?
3. What is “Object Cloning”?

The object cloning is a way to create exact copy of an object. For this purpose, clone () method of Object class is used to clone an object.

The java.lang.Cloneable interface must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone () method generates CloneNotSupportedException.

The clone () method is defined in the Object class. Syntax of the clone() method is as follows:

Why use “Object Cloning”?

The clone() method saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing to be performed that is why we use object cloning.

Advantages of “Object Cloning”?

Less processing task.

As you can see in the above example, both reference variables have the same value. Thus, the clone () copies the values of an object to another. So we don't need to write explicit code to copy the value of an object to another.

If we create another object by new keyword and assign the values of another object to this one, it will require a lot of processing on this object. So to save the extra processing task we use clone () method.

1. Difference of Error and Exception

Error and Exception both extend Throwable, but mostly Error is thrown by JVM in a scenario which is fatal and there is no way for the application program to recover from that error. For instance OutOfMemoryError.

Though even application can raise an Error but it’s just not a good a practice, instead applications should use checked exceptions for recoverable conditions and runtime exceptions for programming errors.

Error is something that most of the time you cannot handle it.

Exception was meant to give you an opportunity to do something with it. Like try something else or write to the log.

try{

//connect to database 1

}

catch(DatabaseConnctionException err){

//connect to database 2

//write the err to log

}

1. Two Types of Exceptions

There are two types of exceptions in java

* + Checked exceptions

Checked exceptions are also known as compile time exceptions as they are identified during compile time of the program. Exception and all the subclasses of Exception class are checked exceptions:

* + Unchecked exceptions

Unchecked Exceptions are also known as runtime exceptions as they are identified during the runtime of the program. Class Runtime Exception and all the subclasses of Runtime Exception are unchecked Exceptions

1. Difference of Checked and Unchecked Exception:

Checked exceptions must be explicitly caught or propagated as described in Basic try-catch-finally Exception Handling. Unchecked exceptions do not have this requirement. They don't have to be caught or declared thrown.

Checked exceptions in Java extend the java.lang.Exception class. Unchecked exceptions extend the java.lang.RuntimeException.

1. Give me Example of Checked Exception
   * **ClassNotFoundException**

This Exception occurs when Java run-time system fail to find the specified class mentioned in the program

* + **InstantiationException**

This Exception occurs when you create an object of an abstract class and interface

* + **IllegalAccessException**

This Exception occurs when you create an object of an abstract class and interface

* + **NotSuchMethodException**

This Exception occurs when the method you call does not exist in class

1. Give me Example of Unchecked Exception
   * **Arithmetic Exception**

These Exception occurs, when you divide a number by zero causes an Arithmetic Exception

* + **Class Cast Exception**

These Exception occurs, when you try to assign a reference variable of a class to an incompatible reference variable of another class

* + **Array Store Exception**

These Exception occurs, when you assign an array which is not compatible with the data type of that array

* + **ArrayIndexOutOfBoundsException**

These Exception occurs, when you assign an array which is not compatible with the data type of that array

* + **Null Pointer Exception**

These Exception occurs, when you try to implement an application without referencing the object and allocating to a memory

* + **Number Format Exception**

These Exception occurs, when you try to convert a string variable in an incorrect format to integer (numeric format) that is not compatible with each other

* + **Negative ArraySizeException**

These are Exception, when you declare an array of negative size.

1. Pros and Cons of Checked VS Unchecked Exception

Pro Checked Exceptions:

Compiler enforced catching or propagation of checked exceptions make it harder to forget handling that exception.

Pro Checked Exceptions:

Unchecked exceptions makes it easier to forget handling errors since the compiler doesn't force the developer to catch or propagate exceptions (reverse of 1).

Pro Unchecked Exceptions:

Checked exceptions that are propagated up the call stack clutter the top level methods, because these methods need to declare throwing all exceptions thrown from methods they call.

Pro Checked Exceptions:

When methods do not declare what unchecked exceptions they may throw it becomes more difficult to handle them.

Pro Unchecked Exceptions:

Checked exceptions thrown become part of a methods interface and makes it harder to add or remove exceptions from the method in later versions of the class or interface.

1. Should there be any specific order in which I should write the following public static void main ()?

We can interchange static and public

Ex.

static public void main(String args[])

static public void main(String... args)

However you cannot reshuffle the return type with any position,

Ex.

public void static main(String[] args) // is wrong

And also

static void public main(String[] args) // is also wrong

You could have easily tried out the various permutations to see what does and does not work. For one thing, none of them will work if you don't change main() to main(String[] args). Beyond that, public and static are modifiers that can come in any order, but most code style conventions have a prescribed order for them anyway. The void must be directly before the method name, since it's the return type, and not a modifier.

1. Why java main method is static?

|  |
| --- |
| Because object is not required to call static method if it were non-static method, jvm create object first then call main () method that will lead the problem of extra memory allocation. |

1. What is a “static” keyword?

The static keyword in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

* **Java static variable**

If you declare any variable as static, it is known static variable.

* The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees, college name of students etc.
* The static variable gets memory only once in class area at the time of class loading.

**Advantage of static variable**

* It makes your program memory efficient (i.e. it saves memory).
* **Java static method**

If you apply static keyword with any method, it is known as static method.

* A static method belongs to the class rather than object of a class.
* A static method can be invoked without the need for creating an instance of a class.
* A static method can access static data member and can change the value of it.

**Restrictions for static method**

* There are two main restrictions for the static method. They are:
* The static method cannot use non static data member or call non-static method directly.
* This and super cannot be used in static context.
* **Java static block**
* Is used to initialize the static data member.
* It is executed before main method at the time of class loading.

1. public static void main(String[] args)

* In this case, main must be declared as public, since it must be called by code outside of its class when the program is started.
* The keyword static allows main to be called without having to instantiate a particular instance of the class.
* The keyword void simply tells the compiler that main does not return a value. The main is the method called when a Java application begins.

1. What is hash?
2. What is hashCode?