Java Basics

Java Environments

http://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqb/index.html

Java Environment consists of:

- Programming Environment
  - A programming language
  - An API specification
  - A virtual machine specification
- Java RunTime Environment
  - JVM
  - Class loader and bytecode verifier
  - Interpreter and garbage collector
  - JIT - Just in time compiler

Install Java

Download JDK

http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html

Set Environments

- JAVA_HOME
- PATH

Java Classpath

- ClassNotFoundException is thrown when the referred Java class is not found by the Java runtime.
- Following are the different ways to add a Java class file or jar file to the Java classpath:
  - Classpath argument in java tool
    - java -classpath "mysql-connector-java-5.1.19-bin.jar" MyJavaClassName
  - Set classpath to multiple jar files:
    - java -cp \tools.jar;.; \utils.jar MyJavaClassName
- Use “ext” location in Java installation directory

Java Package

- Java package is the mechanism to organize the java classes by grouping them.
- The grouping logic can be based on functionality or modules based.
- A java class fully classified name contains package and class name. For example, java.lang.Object is the fully classified name of Object class that is part of java. lang package.
- java.lang package is imported by default and we don’t need to import any class from this package explicitly.

Java Package

- You should bundle these classes and the interface in a package for several reasons, including the following:
  - You and other programmers can easily determine that these types are related.
  - You and other programmers know where to find types that can provide graphics-related functions.
  - The names of your types won't conflict with the type names in other packages because the package creates a new namespace.
  - You can allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package.

Java Package - naming conventions

- Package names are written in all lowercase to avoid conflict with the names of classes or interfaces.
- Companies use their **reversed Internet domain** name to begin their package names - for example, com.microsoft.phone for a package named phone created by a programmer at microsoft.com.
- Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or the project name after the company name (for example, com.company.region.package).
- Packages in the Java language itself begin with **java.** or **javax.**

Java code conventions

http://www.oracle.com/technetwork/java/codeconventions-150003.pdf

NullPointerException

- NullPointerException can be thrown, when there is an attempt to use null where an object is required.
- That is you are trying to access a reference where there is no value.
- Main scenarios are when, calling the instance method of a null object and accessing or modifying the field of a null object.
- When you pass null to a method when it expects a real value. It means to say that null object is used illegally.
- How do we get a null object in place?
    - When you create an object and not initializing it.
    - When you access a method and it returns a null object.

Java Array

- Store same ‘type’ of data that can be logically grouped together.
- Array is a fundamental construct in any programming languages.
- Easy to iterate, easy to store and retrieve using their index.
- In Java, a beginner starts with ***public static void main(String args[])***. The argument for the main method is an array.
- In java arrays are objects.
- Arrays are stored in heap memory.
- When you refer the array values, the index starts from 0 ‘zero’ to ‘n-1′.
- An array index is always a whole number and it can be a int, short, byte, or char.

Java Array

- **ArrayStoreException** – When you try to store a non-compatible value in a java array you will get a ArrayStoreException.
- **Array Declaration**

  int []marks;

  or

  int marks[];

  or

  int[] marks;
- **Array Instantiation**

  marks = new int[5];

Java Array

- Once an array is instantiated, it size cannot be changed.
- All java arrays implements **Cloneable** and **Serializable**.
- **ArrayIndexOutOfBoundsException**
  - You get ArrayIndexOutOfBoundsException when you access an array with an illegal index, that is with a negative number or with a number greater than or equal to its size.
- Iterating a Java Array
- In language C, array of characters is a String but this is not the case in java arrays. But the same behaviour is implemented as a StringBuffer wherein the contents are **mutable**.

Array Copy

- **Copy a Java array**

  Arrays.copyOf(marks, marks.length);

  System.arraycopy(marks, 0, newMarksTwo, 0, marks.length);

Java Primitive

8 primitive data types in Java:

int	32-bit signed two's complement integers
long	64-bit signed two's complement integers
short	16-bit signed two's complement integers
byte	8-bit signed two's complement integers
double	double-precision 64-bit floating point
float	single-precision 32-bit real number
char	single 16-bit Unicode character
boolean	1-bit information(true or false)

Java Primitive - Numeric Types

Numeric types are classified as:

- integral primitive (byte, short, int, long)
- floating point type primitives (float, double)

Note:

- **Two's complement** means, a negative number will be denoted by the two's complement of its absolute value.
- Most significant digit (MSB) will denote if the number is positive or negative.
- MSB will be 0 if the number is positive and 1 if it is negative.

Range

-2 power (N-1) to 2 power (N-1) - 1

where N is the bit size like 8 or 16 ….

Note: The built-in integer operators do not indicate **overflow or underflow** in any way. The only numeric operators that can throw an exception are the integer divide operator/0 and the integer remainder operator %, which throw an ArithmeticException if the right-hand operand is zero.

Integer.MAX_VALUE + 1 == Integer.MIN_VALUE

Integer.MIN_VALUE - 1 == Integer.MAX_VALUE.

Variable

Java variables can be categorized into the following seven types:

1. Class Variable
2. Instance Variable
3. Array Component Variable
4. Method Parameter Variable
5. Constructor Parameter Variable
6. Exception Handler Parameter Variable
7. Local Variable

Heap Memory

- **Heap memory** is used by java runtime to allocate memory to Objects and JRE classes.
- Whenever we create any object, it’s always created in the Heap space.
- Garbage Collection runs on the heap memory to free the memory used by objects that doesn’t have any reference.
- Any object created in the heap space has global access and can be referenced from anywhere of the application.

Stack Memory

- **Stack memory** is used for execution of a thread.
- They contain **method specific values** that are short-lived and references to other objects in the heap that are getting referred from the method.
- Stack memory is always referenced in LIFO (Last-In-First-Out) order.
- Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method.
- As soon as method ends, the block becomes unused and become available for next method.
- Stack memory size is very less compared to Heap memory.

Heap Vs Stack Memory

- Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution.
- Whenever an object is created, it’s always stored in the Heap space and stack memory contains the reference to it. Stack memory only contains local primitive variables and reference variables to objects in heap space.
- Objects stored in the heap are globally accessible whereas stack memory can’t be accessed by other threads.
- Memory management in stack is done in LIFO manner whereas it’s more complex in Heap memory because it’s used globally. Heap memory is divided into Young-Generation, Old-Generation etc.

Heap Vs Stack Memory

- Stack memory is short-lived whereas heap memory lives from the start till the end of application execution.
- We can use **-Xms** and **-Xmx** JVM option to define the startup size and maximum size of heap memory. We can use **-Xss** to define the stack memory size.
- When stack memory is full, Java runtime throws **java.lang.StackOverFlowError** whereas if heap memory is full, it throws **java.lang.OutOfMemoryError**: Java Heap Space error.
- Stack memory size is very less when compared to Heap memory.
- Stack memory is very fast with compared to heap memory.

Pass by value

- One of the biggest confusion in Java programming language is whether it’s Pass by Value or Pass by Reference.
- First of all we should understand what is meant by pass by value or pass by reference.
- **Pass by Value:**
  - The method parameter values are copied to another variable and then the copied object is passed, that’s why it’s called pass by value.
- **Pass by Reference:**
  - An alias or reference to the actual parameter is passed to the method, that’s why it’s called pass by reference.
- **Java is always Pass by Value and not pass by reference.**

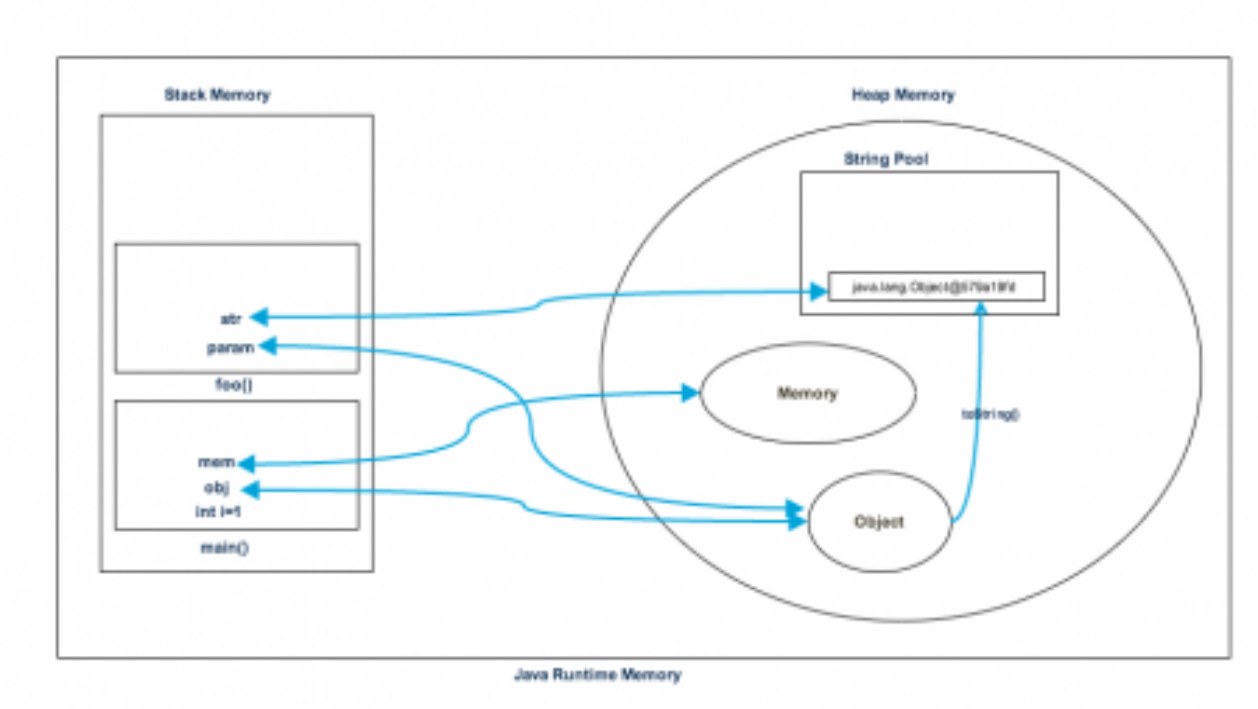
https://www.youtube.com/watch?v=_y7k_0edvuY

Pass by value

Memory.java

```java
package com.journaldev.test;

public class Memory {

    public static void main(String[] args) { // Line 1
        int i=1; // Line 2
        Object obj = new Object(); // Line 3
        Memory mem = new Memory(); // Line 4
        mem.foo(obj); // Line 5
    } // Line 9

    private void foo(Object param) { // Line 6
        String str = param.toString(); //// Line 7
        System.out.println(str);
    } // Line 8

}
```

Pass by value

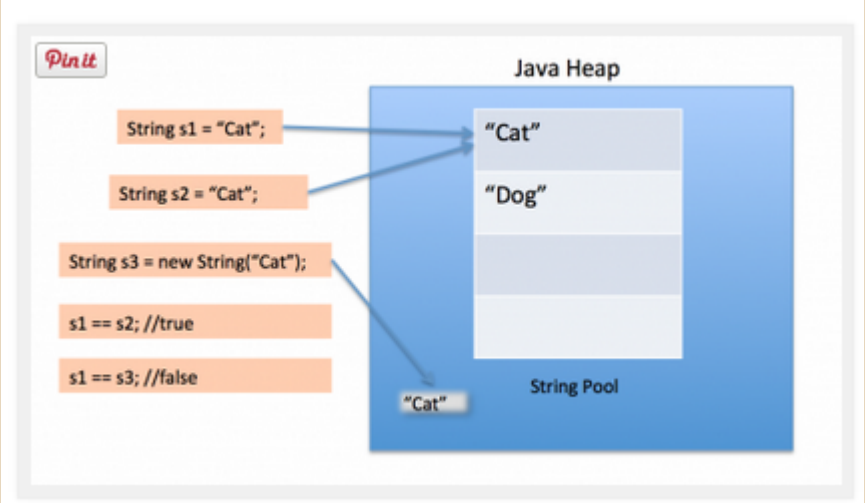


Strong pool

- As the name suggests, **String Pool** is a pool of Strings stored in Java Heap Memory.
- We know that String is special class in java and we can create String object using new operator as well as providing values in double quotes.
- String Pool is possible only because **String is immutable** in Java and it's implementation of String interning concept.
- In computer science, **string interning** is a method of storing only one copy of each distinct string value, which must be immutable
- String pool is also example of Flyweight design pattern.
- String pool helps in saving a lot of space for Java Runtime although it takes more time to create the String.
- When we use double quotes to create a String, it first looks for String with same value in the String pool, if found it just returns the reference else it creates a new String in the pool and then returns the reference.

Strong pool

- Using new operator, we force String class to create a new String object and then we can use intern() method to put it into the pool or refer to other String object from pool having same value.
- When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.



Access Modifiers

- Access modifiers specifies who can access them. There are four access modifiers used in java.

  public, **private**, **protected**, **no modifier**

- Usage of these access modifiers is restricted to two levels.

  class level access modifiers and **member level access modifiers**

- **Class level access modifiers** (java classes only)
  - Only two access modifiers is allowed, **public** and **no modifier**
  - If a class is **'public'**, then it CAN be accessed from ANYWHERE.
  - If a class has **'no modifier'**, then it CAN ONLY be accessed from 'same package'.

Access Modifiers

- **Member level access modifiers** (java variables and java methods)
  - All the four **public**, **private**, **protected** and **no modifier** is allowed.
  - **public and no modifier** - the same way as used in class level.
  - **private** - members CAN ONLY access.
  - **protected** – CAN be accessed from ‘same package’ and a subclass existing in any package can access.

Java Static

- **static** is a keyword in java and we can’t create a class or package name as static.
- **Java static variables:** We can use static keyword with a class level variable.
  - A static variable is a class variable and doesn’t belong to Object/instance of the class.
  - Since static variables are shared across all the instances of Object, they are not thread safe.
  - Usually static variables are used with final keyword for common resources or constants that can be used by all the objects.
  - If the static variable is not private, we can access it with ClassName. variableName

Java Static

- **Java static methods:** Same as static variables, static methods belong to class and not to class instances.
  - A static method can access only static variables of class and invoke only static methods of the class.
  - Usually static methods are utility methods that we want to expose to be used by other classes without the need of creating an instance; for example Collections class.
  - Java Wrapper classes and utility classes contains a lot of static methods.
  - The main() method that is the entry point of a java program itself is a static method.

Java Static

- From Java 8 onwards, we can have static methods in interfaces too
- **Java static Block:**
  - Java static block is the group of statements that gets executed when the class is loaded into memory by Java ClassLoader.
  - It is used to initialize static variables of the class.
  - Mostly it’s used to create static resources when class is loaded.
  - We can’t access non-static variables in static block.
  - We can have multiple static blocks in a class, although it doesn’t make much sense.
  - Static block code is executed only once when class is loaded into memory.

Java Static

- **Java static class:**
  - We can use static keyword with nested classes.
  - static keyword can’t be used with top-level classes.
  - Static nested class is same as any other top-level class and is nested for only packaging convenience.

Final Keyword

- A java variable can be declared using the keyword **final**. Then the final variable can be assigned only once.
- A variable that is declared as final and not initialized is called a **blank final variable**. A blank final variable forces the constructors to initialise it.
- Java classes declared as final cannot be extended. **Restricting inheritance!**
- Methods declared as final cannot be **overridden**. In methods private is equal to final, but in variables it is not.

Final Keyword

- **final parameters** – values of the parameters cannot be changed after initialization. Do a small java exercise to find out the implications of final parameters in method overriding.
- Java local classes can only reference local variables and parameters that are declared as final.
- A visible advantage of declaring a java variable as static final is, the compiled java class results in faster performance.

Inheritance

- Inheritance is one of the basic concepts of Object Oriented Programming.
- Java supports inheritance and define an **is-a** relationship between a superclass and a subclass.
- For example if Vehicle is the superclass and Car is the subclass inherited from superclass Vehicle, then **Car is-a Vehicle** too.
- Inheritance in Java is the method to create a new class from an existing class.
- Inheritance in java is **transitive**, so if Santo extends class Car, then Santro is also inherited from Vehicle class. Vehicle becomes the superclass of both Car and Santro.
- In java, every class implicitly extends java.lang.Object class, so Object class is at the top level of inheritance hierarchy in java.

Inheritance

- Private members of superclass are not directly accessible to subclass but these can be indirectly accessible via getter and setter methods.
- Superclass constructors are not inherited by subclass.
- Java doesn’t support multiple inheritance, a subclass can extends only one class.
- So Vehicle is implicitly extending Object class and Car is extending Vehicle class but due to java inheritance transitive nature, Santro class also extends Object class.

Interface

- Interfaces are core part of java programming language and used a lot not only in JDK but also java **design patterns**, most of the **frameworks** and **tools**.
- Interfaces provide a way to **achieve abstraction** in java and used to define the **contract** for the subclasses to implement.
- **interface** is the code that is used to create an interface in java.
- We can’t instantiate an interface in java.
- Interface provides absolute abstraction. Interface only define method definition.
- Interfaces can’t have constructors because we can’t instantiate them and interfaces can’t have a method with body.

Interface

- By default any attribute of interface is public, static and final, so we don’t need to provide access modifiers to the attributes but if we do, compiler doesn’t complain about it either.
- By default interface methods are implicitly **abstract and public**, it makes total sense because the method don’t have body and so that subclasses can provide the method implementation.
- An interface can’t extend any class but it can extend another interface.
- **implements** keyword is used by classes to implement an interface.
- A class implementing an interface must provide implementation for all of its method unless it’s an abstract class.

Interface - Benefits

- Interface provides a contract for all the implementation classes, so its good to code in terms of interfaces because implementation classes can’t remove the methods we are using.
- Interfaces are good for starting point to define Type and create top level hierarchy in our code.
- Since a java class can implements multiple interfaces, it’s better to use interfaces as super class in most of the cases.

Abstract Class

- A Java class that is declared using the keyword abstract is called an **abstract** class.
- **New instances cannot be created** for an abstract class but it can be **extended**.
- An abstract class can have abstract methods and concrete methods or both.
- Methods with implementation body are concrete methods.
- An abstract class can have static fields and methods and they can be used the same way as used in a concrete class.
- If abstract class doesn’t have any method implementation, its better to use interface because java doesn’t support multiple class inheritance.

Abstract Method

- A method that is declared using the keyword abstract is called an abstract method.
- Abstract methods are declaration only and it will not have implementation.
- It will not have a method body.
- A Java class containing an abstract method must be declared as abstract class.
- An abstract method can only set a visibility modifier, one of public or protected.
- That is, an abstract method cannot add static or final modifier to the declaration.

Java Cast

- Identity conversions
- Widening primitive conversions
- Narrowing primitive conversions
- Widening reference conversions
- Narrowing reference conversions
- Boxing conversions
- Unboxing conversions
- Unchecked conversions
- Capture conversions
- String conversions
- Value set conversions

Widening Primitive Conversion

- **A widening primitive conversion does not lose information about the overall magnitude of a numeric value.**
- There is no cast required and will never result in a runtime exception.
- Following are the possible widening conversions:
  - byte to short, int, long, float, or double
  - short to int, long, float, or double
  - char to int, long, float, or double
  - int to long, float, or double
  - long to float or double
  - float to double

Narrowing Primitive Conversion

- **A narrowing primitive conversion may lose information about the overall magnitude of a numeric value and may also lose precision and range.**
- Cast required between types.
- Overflow and underflow may happen, but no runtime exception.
- Following are the possible narrowing conversions:
  - short to byte or char
  - char to byte or short
  - int to byte, short, or char
  - long to byte, short, char, or int
  - float to byte, short, char, int, or long
  - double to byte, short, char, int, long, or float

Reference Conversions and Cast

- Casting a Float into Integer is not proper and will get compile error as inconvertible types. **Casting in Java is done within same hierarchy of types, that is between inherited types.**
- **upcast -** Casting a subtype object into a supertype and this is called upcast. Need not add an explicit cast and you can assign the object directly.
- **downcast -** Casting a supertype to a subtype is called downcast. This is the mostly done cast.
- **ClassCastExcpetion -** We get ClassCastException in a downcast. During runtime, because of unforeseen circumstances, the value is not of expected subtype. In such cases, we get ClassCastException.

Boxing and Unboxing

- Boxing and Unboxing Conversions
- Converting from a primitive type to its corresponding reference type is boxing conversion and vice versa is unboxing conversion.
- **Examples:**
  - From primitive boolean to type Boolean - BOXING
  - From reference type Boolean to primitive boolean - UNBOXING
  - From primitive int to type Integer - BOXING
  - From reference type Integer to primitive int - UNBOXING

String Conversion

- String conversion applies only to the ‘+’ operator, when one operand is a String and another is a primitive type.
- In such a case, primitive type is converted to its corresponding reference type and then it is converted using the toString() method.
- No cast is required.

this operator

1. To specifically denote that the instance variable is used instead of static or local variable.

```
private String javaFAQ;

void methodName(String javaFAQ) {
    this.javaFAQ = javaFAQ;
}
```

- Here **this** refers to the instance variable. Here the precedence is high for the local variable.
- The absence of the “**this**” denotes the local variable.

this operator

2. **this** is used to refer the constructors

```
// 2.
public Person(String firstName, String lastName, String empId) {
    this(firstName, lastName);
    this.empId = empId;
    display(this);
}

// 3.
public void display(Person person) {
    System.out.println("firstName: " + person.getFirstName() + " lastName: " + person.getLastN
            + person.getEmpId());
}
```

3. This is used to pass the current java instance as parameter

4. Similar to the above, java this can also be used to return the current instance(**return this**)

5. Java This can be used to get the handle of the current class (**this.getClass()**)

marker interface

“An interface is called a marker interface when it is provided as a handle by Java interpreter to mark a class so that it can provide special behaviour to it at runtime and they do not have any method declarations”.

- We cannot create marker interfaces, as you cannot instruct JVM to add special behavior to all classes implementing (directly) that special interface.
  - java.lang.Cloneable
  - java.io.Serializable
  - java.util.EventListener
- From java 1.5, the need for marker interface is eliminated by the introduction of the **java annotation** feature. So, it is wise to use java annotations than the marker interface. It has more feature and advantages than the java marker interface.

Annotations

- Annotation is **code about the code**, that is metadata about the program itself.
- Organized data about the code, embedded within the code itself.
- Parsed by the annotation parsing tool or by compiler and can also be made available at run-time too.
- Annotations are far more powerful than java comments and javadoc comments.
- Annotations provide a generic way of adding information to class/method/field/variable.
- This avoids human errors and decreases development time as always with automation.
- Frameworks like Hibernate, Spring, Jersey make heavy use of annotations.

Custom Annotations

- Creating custom annotation is similar to writing an interface, except that interface keyword is prefixed with @ symbol.
- We attach ‘@’ just before interface keyword.
- Methods will not have parameters.
- Methods will not have throws clause.
- Method return types are restricted to primitives, String, Class, enums, annotations, and arrays of the preceding types.
- We can assign a default value to method.

Meta Annotation

- Annotations itself is meta information then what is meta annotations?
- It is information about annotation.
- When we annotate a annotation type then it is called meta annotation.
- For example, we say that this annotation can be used only for methods.
    - @Target(ElementType.METHOD)
    - public @interface MethodInfo { }

Annotation Usage

- To describe constraints or usage of an element: e.g. @Deprecated, @Override, or @NotNull
- To describe the "nature" of an element, e.g. @Entity, @TestCase, @WebService
- To describe the behavior of an element: @Statefull, @Transaction
- To describe how to process the element: @Column, @XmlElement

Annotation Types

- **Documented**
  - Wherever this annotation is used those elements should be documented using Javadoc tool.
- **Inherited**
  - This meta annotation denotes that the annotation type can be inherited from superclass. Super class will be queried till a matching annotation is found.
- **Target**
  - Annotation type is applicable for only the element (ElementType) listed. CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE.

    ```
    @Target(ElementType.FIELD)
    public @interface FieldInfo { }
    ```

Annotation Types

- **Retention**
  - When an annotation type is annotated with meta annotation **Retention**, **RetentionPolicy** has three possible values:

    ```
    @Retention(RetentionPolicy.RUNTIME)
    public @interface Developer {
        String value();
    }
    ```

  - **Class:**
    - When the annotation value is given as ‘class’ then this annotation will be compiled and included in the class file.

Annotation Types

- **Runtime**
  - The value name itself says, when the retention value is **‘Runtime’** this annotation will be available in JVM at runtime.
  - We can write custom code using reflection package and parse the annotation.
- **Source**
  - This annotation will be removed at compile time and will not be available at compiled class.

Variable Argument

- Java variable arguments was **introduced in Java 5** and also known as **java varargs**.
- It enables a method to accept variable number of arguments.
- We use three dots (…) also known as **Ellipsis** in the method signature to make it accept variable arguments.
- We can have only one varargs in the method.
- Only the last argument of a method can be varargs.
- According to java documentation, we should not overload a varargs method.
- It is also known as '**variable arity method**' and '**variable arity parameter**' in java language specification.
- It is not recommended to overload **varargs** methods.

Variable Argument - How it works ?

- Java compiler removes **elipsis (…)** and replaces it with an array in bytecode.
- JVM is not aware of varargs and so when we use reflection there is no provision to pass variable arguments and we need to construct an array and pass it.
- Example of JDK usage:
  - java.lang.Class#getMethod
  - java.io.PrintStream#printf

String

- String class represents character strings, we can instantiate String by two ways.
  - String str = “abc”; or
  - String str = new String (“abc”);
- String is **immutable** in java, so its easy to share it across different threads or functions.
- When we create a String using double quotes, it first looks for the String with same value in the JVM **string pool**, if found it returns the reference else it creates the String object and then place it in the String pool.
- This way JVM saves a lot of space by using same String in different threads.
- But if new operator is used, it explicitly creates a new String in the heap memory.
- + operator is overloaded for String and used to concatenate two Strings. Although internally it uses StringBuilder to perform this action.

String

- String overrides **equals()** and **hashCode()** methods, two Strings are equal only if they have same characters in same order.
- Note that equals() method is case sensitive, so if you are not looking for case sensitive checks, you should use **equalsIgnoreCase()** method.
- A String represents a string in the **UTF-16** format
- String is a final class with all the fields as final except “private int hash”. This field contains the hashCode() function value and created only when hashCode() method is called and then cached in this field.
- Hash is generated using final fields of String class with some calculations, so every time hashCode() method is called, it will result in same output.
- For caller, its like calculations are happening every time but internally it’s cached in hash field.

String vs StringBuffer

- These are heavy operations and generate a lot of garbage in heap. So Java has provided **StringBuffer** and **StringBuilder** class that should be used for String manipulation.
- StringBuffer and StringBuilder are **mutable** objects in java and provide append(), insert(), delete() and substring() methods for String manipulation.

StringBuilder vs StringBuffer

- **StringBuffer** was the only choice for String manipulation till Java 1.4 but it has one disadvantage that all of its public methods are synchronized, it provides Thread safety but on a performance cost.
- Java 1.5 introduced a new class StringBuilder that is similar with StringBuffer except thread safety and synchronization.
- If you are in a single threaded environment or don’t care about thread safety, you should use StringBuilder else use StringBuffer.

enums

- Enum was introduced in Java 1.5 as a new type whose fields consists of fixed set of constants.
- For example, in Java we can create Direction as enum with fixed fields as EAST, WEST, NORTH, SOUTH.
- Use of Enum valueOf, enum values, EnumSet and EnumMap with examples.
- **enum** is the keyword to create an enum type and similar to class.
- enum is better than normal constants fields in java classes.
- We have two risks with using constants that are solved by enum.
    - We can pass any value to the constant but we can pass only fixed values to enums, so it provides **type safety.**
    - We can change the constants value but if we change the enum constants, we will get compile time exception which removes any possibility of runtime issues.

enums

- Enum was introduced in Java 1.5 as a new type whose fields consists of fixed set of constants.
- For example, in Java we can create Direction as enum with fixed fields as EAST, WEST, NORTH, SOUTH.
- Use of Enum valueOf, enum values, EnumSet and EnumMap with examples.
- **enum** is the keyword to create an enum type and similar to class.
- enum is better than normal constants fields in java classes.
- We have two risks with using constants that are solved by enum.
    - We can pass any value to the constant but we can pass only fixed values to enums, so it provides **type safety.**
    - We can change the constants value but if we change the enum constants, we will get compile time exception which removes any possibility of runtime issues.