Java Notes

Sunday, February 11, 2018 8:43 AM

**** JAVA Program**
A Java program is mostly a collection of objects talking to other objects by invoking each other's methods. Every object is of a certain type, and that type is defined by a class or an interface.

**** JAVA BASICS**

1. **Sequence of source**

   License and copyright
   package Declaration
   Import Declaration
   Class Declaration
   Variable Declaration
   Method Declaration

## All Method you need to access outside you need to mark it public.

**** Class Definition**
Class A template that describes the kinds of state and behavior that objects of its type support.

■ **Object** At runtime, when the Java Virtual Machine (JVM) encounters the new keyword, it will use the appropriate class to make an object which is an instance of that class. That object will have its own state, and access to all of the behaviors defined by its class.

■ **State** (instance variables) Each object (instance of a class) will have its own unique set of instance variables as defined in the class. Collectively, the values assigned to an object's instance variables make up the object's state.

■ **Behavior** (methods) When a programmer creates a class, she creates methods for that class. Methods are where the class' logic is stored. Methods are where the real work gets done. They are where algorithms get executed, and data gets manipulated.

## There can be only one public class per source code file.
## If there is a public class in a file, the name of the file must match the name of the public class. For example, a class declared as public class Dog { } must be in a source code file named Dog.java.

**** Abstract Class**
## never use final and abstract it will not compile.
## If single method is abstract class should be Abstract Class.
## Never use private with abstract because its need to be visible.

Examples:
1. basic Rules

```
 package com.dev.classDev;

/**
 *
 * @author rkumar
 * Abstract Demo Class
 * 1. Abstract methods should not be final because it should be visible
 *
 */
public abstract class AbstractDemo {

	public abstract void show();

	abstract void go();

}
```

2. **Important Rule**

```
	package com.dev.classDev;

	/**
	 *
	 * @author rkumar
	 * Another Abstract Method
	 * 1. Abstract class can extends another class and should not be force to implement all the
	function
	 *
	 */
	public abstract class AbstractDemo1 extends AbstractDemo {
		public abstract void go();
		public abstract void show1();

	}
```

3. Abstract Implementation

```
	package com.dev.classDev;

	/**
	 *
	 * @author rkumar
	 * Abstract implementation class
	 * 1. Sub class need to implement all the function of Abstract class
	 * 2. If Abstract class extends another abstract class then that class
	 *    functions also need to be implemented in subclass
	 * 3. We can't create Object of Abstract Class but We can instantiate through Sub Class or use this
	abstract
	 *   methods in subclass
	 */
	public class AbstractImplementation extends AbstractDemo1 {

		@Override
		public void go() {
			// TODO Auto-generated method stub
	}

	@Override
	public void show1() {
		// TODO Auto-generated method stub

	}

	@Override
	public void show() {
		// TODO Auto-generated method stub

	}
}
```

**** Interface**

Rules:

1. Methods will be always public and abstract(not required to explicitly add that).
2. All variable defined interface must be public, static, final means line constant(not required to explicitly add that).
3. Interface method must not be static .
4. Because interface methods are abstract, it can't be marked final.
5. Interface always must be public static final.
6. Basic rules
   a. Interface can extends another interface
   b. Class can implement interface not extends
   c. Implementation class should implement all the methods by defaults

Examples:

1. **Basic**

```
package com.dev.interfaceDemo;

/**
 *
 * @author rkumar
 * Demo interface for method and variable
 * 1. Interface can extends another interface
 * 2. Class can implement interface not extends
 * 3. Implementation class should implement all the methods by defaults
 *
 */
public interface InterfaceDemo {

	// variable always by default public, static and final
	String name = "Ritesh Kumar";

	// methods always by default public and abstract
	String getName();

}
```

2. Implementation

```
package com.dev.interfaceDemo;

/**
 *
 * @author rkumar
 *         Implementation class for interface demo
 */
public class InterfaceDemoImplemenation implements InterfaceDemo {

        @Override
        public String getName() {
//                      name = "kumar"; /* final interface varibale can't be assigned */
                return name;
        }

        // Main Methods
        public static void main(String args[]) {
                InterfaceDemoImplemenation demo = new InterfaceDemoImplemenation();
                System.out.println("name: "+demo.getName());
        }

}
```

**** Access Modifiers**

## In practice it's nearly almost best keep all variable private or protected. If variables needs to be change set or read, programmer should use public accessor method rather than directly access.

Types of access modifier:

1. Protected: - This make the variables accessible to all class inside the certification package as well as inheritable by subclass outside this package.
2. Default:- Package Level
3. Public :- Global Access
4. Private: - Class member access

## There is never a case where an access modifier can be applied to local variable.
## You can be certain that any local variable declared with an access modifier's will not compile. There is only one modifier that can be applied to local variable is final.

Imp Table:

Visibility	**Public**	**Protected**	**Default**	**Private**
From the same Class	yes	yes	yes	yes
From any class in Same package	yes	yes	yes	no
From a subclass in Same package	yes	yes	yes	no
From a subclass Outside the same	yes	yes through inheritance	no	no

Package				
From any non-subclass Outside the package	yes	no	no	no

**** Final** :

Variable - Can't be change (Constant)
method - not override
class - not subclass

**** Native Method:**
The native modifiers indicates that a method is implemented in platform dependent code in C. Native is a modifiers not a class, not a variable, just method.
Native method's body must be a semicolon like abstract method.

**** Strictfp Methods**
With Strictfp, You can predict how your floating point will behave regardless of the underlying platform the JVM is running on. The downside is that if the underlying platform is capable of supporting greater precision, a sctrictfp method won't be able to take advantage of it.

**** Argument and Parameter:**

When you pass original value is called argument.
doStuff("a", 2);

The thing in the methods signature that indicates what must receive when it's invoked.
Void doStuff(String a, int a);

We are expecting two parameters string, int

**** Var-args**

Void doStuff(int ...x) kind of array this needs to always last argument in methods.

**** Object Orientation:**

**** Encapsulation :**
The ability to make changes in your implementation code without breaking the code of others who use your code is a key benefit of encapsulation.

If you want maintainability, flexibility, and extensibility (and of course, you do), your design must include encapsulation. How do you do that?
■ Keep instance variables protected (with an access modifier, often private).
■ Make public accessor methods, and force calling code to use those methods rather than directly accessing the instance variable.
■ For the methods, use the JavaBeans naming convention of set<someProperty> and get<someProperty>.

If We are Using setter and getter method implementation then we can validate the variable before setting it into variable like

If some one sending negative value to set so we can discard the negative value so it will return default value;

Example:

```
package com.practice.ooo;

/**
 * class to demo Encapsulation
 *
 * @author rkumar
 *
 */
public class EncapsulationDemo {
	// Data Hiding
	// private variables
	private String name = "Ritesh";
	private int rollNo = 1;

	// setter and getter methods
	public void setName(String name) {
		this.name = name;
	}

	public String getName() {
		return name;
	}

	public void setRollNo(int rollNo) {
		// Added validation for negative number
		if (rollNo >= 0) {
			this.rollNo = rollNo;
		}

	}

	public int getRollNo() {
		return rollNo;
	}
}

/**
 * class to test encapsulation
 *
 * @author rkumar
 *
 */
class test {
	public static void main(String args[]) {
		EncapsulationDemo encapsulationDemo = new EncapsulationDemo();
		encapsulationDemo.setName("Kumar");
		encapsulationDemo.setRollNo(-1);
		System.out.println("name : " + encapsulationDemo.getName());
```

```
                // It returns 1 because setRollNo doesn't support negative number which can't achieve
                // if variable declares publc
                System.out.println("rollNo : " + encapsulationDemo.getRollNo());
        }
}
```

***** Inheritance:**

 ## Instanceof returns true if the reference variable being tested is of type being compared to.

## Every class in java is a subclass of Object, (Except of course class Object itself).

## Common reason to use inheritance in java
   1. To promote code reuse.
   2. To use polymorphism.

1. A common design approach is to create a fairly generic version of a class with the intention of creating more specialized subclasses that inherit from it. For example:

```
package com.practice.ooo;

/**
 * Super class for demo
 * @author rkumar
 *
 */
class Car {
        public void color() {
                System.out.println("Generic color: ");
        }
}

/**
 * Sub calss for Car
 * @author rkumar
 *
 */
class Audi extends Car {
        public void speed() {
                System.out.println("Audi Speed Demo: ");
        }
}

/**
 * Main class to test inheritance demo
 * @author rkumar
 *
 */
public class InhertanceDemo {
        // test method for inheritance
        public static void main(String args[]) {
                Audi audi = new Audi();
                audi.color();
```

```
                    audi.speed();
          }
}
```
## Code reuse through inheritance means that methods with generic functionality (like display())—that could apply to a wide range of different kinds of shapes in a game—don't have to be re implemented. That means all specialized subclasses of GameShape are guaranteed to have the capabilities of the more generic
superclass. You don't want to have to rewrite the display() code in each of your specialized components of an online game.

Example :

```
package com.practice.ooo;

/**
 *
 * @author rkumar
 *
 */
class CarDemo{

          public void showColor() {
                    System.out.println(" Color of the car");
          }

          public void shape() {
                    System.out.println("shape of car");
          }
}

class Alto extends CarDemo {
          public void getPrice() {
                    System.out.println("price of alto");
          }

          public void showColor() {
                    System.out.println("Color of the alto");
          }

}

class Maruti extends CarDemo {
          public void getPrice() {
                    System.out.println("price of alto");
          }

          public void showColor() {
                    System.out.println("Color of the suzuki");
          }
}

public class InheritanceDemo2 {
          static void display(CarDemo car) {
```

```
		car.showColor();
	}

	public static void main(String args[]) {

		Alto alto = new Alto();
		Maruti maruti = new Maruti();
		CarDemo carDemo = new Maruti();
		display(alto); //  Color of the alto
		display(maruti); //  Color of the suzuki
		display(carDemo); // Colour of the suzuki
		carDemo.shape(); //  shape of the car

	}
}
```

## Polymorphic method invocations apply only to instance methods. You can always refer to an object with a more general reference variable type (a superclass or interface), but at runtime, the ONLY things that are dynamically selected based on the actual object (rather than the reference type) are instance methods. Not static methods. Not variables. Only overridden instance methods are dynamically invoked based on the real object's type.

***** IS-A and HAS-A Relationships**

IS-A
In OO, the concept of IS-A is based on class inheritance or interface implementation. IS-A is a way of saying, "this thing is a type of that thing." For example, a Mustang is a type of horse, so in OO terms we can say, "Mustang IS-A Horse." Subaru IS-A Car. Broccoli IS-A Vegetable (not a very fun one, but it still counts). You express the IS-A relationship in Java through the keywords extends (for class inheritance) and implements (for interface implementation).

Example:
Returning to our IS-A relationship, the following statements are true:
"Car extends Vehicle" means "Car IS-A Vehicle."
"Subaru extends Car" means "Subaru IS-A Car."

"Subaru IS-A Vehicle" because a class is said to be "a type of" anything further upin its inheritance tree.

Subaru extends Car
Car extends Vehicle
So Subaru extends Vehicle means Subaru IS-A Vahicle

HAS-A

HAS-A relationships are based on usage, rather than inheritance. In other words, class A HAS-A B if code in class A has a reference to an instance of class B. For example, you can say the following,

```
package com.practice.ooo;

class Animal {}
class Halter {
	public void tieRope(String rope) {
```

```
                System.out.println(rope);
        }
}
class Horse extends Animal {
        private Halter halter = new Halter();
        public void tie() {
                halter.tieRope("Rope tied");
        }
}
public class HasADemo {
        public static void main(String args[]) {
                Horse horse = new Horse();
                horse.tie();
        }
}
```

***** Polymorphism:**

## Any Java object that can pass more than one IS-A test can be considered polymorphic. Other than objects of type Object, all Java objects are polymorphic in that they pass the IS-A test for their own type and for class Object. Remember that the only way to access an object is through a reference variable, and there are a few key things to remember about references:

■ A reference variable can be of only one type, and once declared, that type can never be changed (although the object it references can change).
■ A reference is a variable, so it can be reassigned to other objects, (unless the reference is declared final).
■ A reference variable's type determines the methods that can be invoked on the object the variable is referencing.
■ A reference variable can refer to any object of the same type as the declared reference, or—this is the big one—it can refer to any subtype of the declared type!
■ A reference variable can be declared as a class type or an interface type. If the variable is declared as an interface type, it can reference any object of any class that implements the interface.

## class PlayerPiece extends GameShape, Animatable -- NO . A class cannot extend more than one class in Java.

## Some languages (like C++) allow a class to extend more than one other class.This capability is known as "multiple inheritance."

***** Override**

The rules for overriding a method are as follows:

■ The argument list must exactly match that of the overridden method. If they don't match, you can end up with an overloaded method you didn't intend.
■ The return type must be the same as, or a subtype of, the return type declared in the original overridden method in the superclass. (More on this in a few pages when we discuss covariant returns.)
■ The access level can't be more restrictive than the overridden method's.
■ The access level CAN be less restrictive than that of the overridden method.

■ Instance methods can be overridden only if they are inherited by the subclass.
A subclass within the same package as the instance's superclass can override
any superclass method that is not marked private or final. A subclass in a
different package can override only those non-final methods marked public
or protected (since protected methods are inherited by the subclass).
■ The overriding method CAN throw any unchecked (runtime) exception,
regardless of whether the overridden method declares the exception. (More
in Chapter 5.)
■ The overriding method must NOT throw checked exceptions that are new
or broader than those declared by the overridden method. For example, a
method that declares a FileNotFoundException cannot be overridden by a
method that declares a SQLException, Exception, or any other non-runtime
exception unless it's a subclass of FileNotFoundException.
■ The overriding method can throw narrower or fewer exceptions. Just because
an overridden method "takes risks" doesn't mean that the overriding subclass'
exception takes the same risks. Bottom line: an overriding method doesn't
have to declare any exceptions that it will never throw, regardless of what the
overridden method declares.
■ You cannot override a method marked final.
■ You cannot override a method marked static. We'll look at an example in a
few pages when we discuss static methods in more detail.
■ If a method can't be inherited, you cannot override it.

***** Overload**

Overloaded methods let you reuse the same method name in a class, but with
different arguments (and optionally, a different return type). Overloading a method
often means you're being a little nicer to those who call your methods, because your
code takes on the burden of coping with different argument types rather than forcing
the caller to do conversions prior to invoking your method. The rules are simple:
■ Overloaded methods MUST change the argument list.
■ Overloaded methods CAN change the return type.
■ Overloaded methods CAN change the access modifier.
■ Overloaded methods CAN declare new or broader checked exceptions.
■ A method can be overloaded in the same class or in a subclass. In other words,
if class A defines a doStuff(int i) method, the subclass B could define a
doStuff(String s) method without overriding the superclass version that
takes an int. So two methods with the same name but in different classes
can still be considered overloaded, if the subclass inherits one version of the
method and then declares another overloaded version in its class definition.

Overload : Argument -must change, return type - can change, Exception - can change
Invocation - Compile time

Override: Argument -must not change, return type - can't change except covariant return, Exception - can reduce or eliminate must not throw new or border checked exception.
Invocation - Run time

***** Interface**

Interface implementation class rule

In order to be a legal implementation class, a
nonabstract implementation class must do the following:
■ Provide concrete (nonabstract) implementations for all methods from the
declared interface.
■ Follow all the rules for legal overrides.
■ Declare no checked exceptions on implementation methods other than
those declared by the interface method, or subclasses of those declared by
the interface method.
■ Maintain the signature of the interface method, and maintain the same
return type (or a subtype). (But it does not have to declare the exceptions
declared in the interface method declaration.)

## Rules for interface

1. A class can implement more than one interface. It's perfectly legal to say, for example, the following:
public class Ball implements Bounceable, Serializable, Runnable
{ ... }
2. An interface can itself extend another interface, but never implement anything. The following code is perfectly legal:
public interface Bounceable extends Moveable { } // ok!

Example:

```
package com.practice.ooo;

interface Demo1 {
	void show();
}

interface Demo2 {
	void print();
}

interface Demo3 extends Demo1, Demo2 {
	// Not required to implement all other interface methods if interface extends other interface
}
public class InterfaceDemo implements Demo3{

	@Override
	public void show() {
		// TODO Auto-generated method stub

	}

	@Override
	public void print() {
		// TODO Auto-generated method stub

	}

}
```

***** Constructor Basic**

## Every class, including abstract classes, MUST have a constructor. Burn that into your brain. But just because a class must have one, doesn't mean the programmer has to type it.

Two key points to remember about constructors are that they have no return type and their names must exactly match the class name.

## Constructor chaining :

If you creating object for subclass, all super class constructor also will be initialized.
Example:
Horse h = new Horse();

1. Main() calls new Horse()
2. Horse() calls super()
3. Animal calls super()
4. Object()

## Rules to Constructor:
■ Constructors can use any access modifier, including private. (A private constructor means only code within the class itself can instantiate an object of that type, so if the private constructor class wants to allow an instance of the class to be used, the class must provide a static method or variable that allows access to an instance created from within the class.)
■ The constructor name must match the name of the class.
■ Constructors must not have a return type.
■ It's legal (but stupid) to have a method with the same name as the class, but that doesn't make it a constructor. If you see a return type, it's a method rather than a constructor. In fact, you could have both a method and a constructor with the same name—the name of the class—in the same class, and that's not a problem for Java. Be careful not to mistake a method for a constructor—be sure to look for a return type.
■ If you don't type a constructor into your class code, a default constructor will be automatically generated by the compiler.
■ The default constructor is ALWAYS a no-arg constructor.
■ If you want a no-arg constructor and you've typed any other constructor(s) any other constructor) for you. In other words, if you've typed in a constructor with arguments, you won't have a no-arg constructor unless you type it in yourself!
■ Every constructor has, as its first statement, either a call to an overloaded constructor (this()) or a call to the superclass constructor (super()), although remember that this call can be inserted by the compiler.
■ If you do type in a constructor (as opposed to relying on the compiler-generated default constructor), and you do not type in the call to super() or a call to this(), the compiler will insert a no-arg call to super() for you, as the very first statement in the constructor.
■ A call to super() can be either a no-arg call or can include arguments passed to the super constructor.
■ A no-arg constructor is not necessarily the default (i.e., compiler-supplied) constructor, although the default constructor is always a no-arg constructor. The default constructor is the one the compiler provides! While the default constructor is always a no-arg constructor, you're free to put in your own noarg constructor.
■ You cannot make a call to an instance method, or access an instance variable,

until after the super constructor runs.
■ Only static variables and methods can be accessed as part of the call to super() or this(). (Example: super(Animal.NAME) is OK, because NAME is declared as a static variable.)
■ Abstract classes have constructors, and those constructors are always called when a concrete subclass is instantiated.
■ Interfaces do not have constructors. Interfaces are not part of an object's inheritance tree.
■ The only way a constructor can be invoked is from within another constructor.

## How do you know what the default constructor will look like? Because...
■ The default constructor has the same access modifier as the class.
■ The default constructor has no arguments.
■ The default constructor includes a no-arg call to the super constructor (super()).

## So if your super constructor (that is, the constructor of your immediate superclass/parent) has arguments, you must type in the call to super(), supplying the appropriate arguments. Crucial point: if your superclass does not have a no-arg constructor, you must type a constructor in your class (the subclass) because you need a place to put in the call to super with the appropriate arguments.
The following is an example of the problem:

```
package com.practice.ooo;

class Const1 {

    Const1(String name) {
        // TODO Auto-generated constructor stub
    }

    // Need to add this for if want to execute this code
    public Const1() {
        // TODO Auto-generated constructor stub
    }
//
    void show() {
        System.out.println("Method to show super class show method");
    }
}

class Const2 extends Const1 { // not compile because it requires default constructor in super clas
    void show() {
        System.out.println("Method of subclass show method");
    }
}
/**
 *
 * @author rkumar
 * Demo Class to show constructor and object initialization with inheritance
 */
public class ConstructorSuperDemo {
    /**
     *
     * @param args
     * Test method to check for constructor initialization
     */
    public static void main(String args[]) {
        Const2 const2 = new Const2();
        const2.show();
    }
}
```

## constructors are never inherited

## Overload Constructor
Overloading a constructor means typing in multiple versions of the constructor, each having a different argument list, like the following examples:

```
class Foo {
    Foo(){}
    Foo(String s){}
}
```

## Static Variable and Methods:
One of the mistakes most often made by new Java programmers is attempting to access an instance variable (which means nonstatic variable) from the static main() method (which doesn't know anything about any instances, so it can't access the variable). The following code is an example of illegal access of a nonstatic
variable from a static method:
Understand that this code will never compile, because you can't access a nonstatic (instance) variable from a static method. Just think of the compiler saying, "Hey, I have no idea which Foo object's x variable you're trying to print!" Remember, it's the class running the main() method, not an instance of the class. Of course, the tricky part for the exam is that the question won't look as obvious as the preceding code. The problem you're being tested for— accessing a nonstatic variable from a static method—will be buried in code that might appear to be testing something else. For example, the preceding code would be more likely to appear as So while you're trying to follow the logic, the real issue is that x and y can't be used within main(), because x and y are instance, not static, variables! The same applies for accessing nonstatic methods from a static method. The rule is, a static method
of a class can't access a nonstatic (instance) method or variable of its own class.

```
package com.practice.ooo;

/**
 *
 * @author rkumar
 * Class to show static use case
 *
 */
public class StaticDemo {
    static int frogCount = 0;
    // int frogCount = 0;
    public StaticDemo() {
        frogCount += 1;
    }

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

```
                new StaticDemo();
                new StaticDemo();
                new StaticDemo();
                new StaticDemo();
                // incase of static frogcount is 4
                // non static varible show error like nonstatic variable frogCount
                // cannot be referenced from a static context
                System.out.println("Frog count : "+frogCount);

        }
}
```

static methods can't be overridden.

***** Coupling and Cohesion:**

Good design has tight encapsulation, loose coupling and high cohesion in classes.

The most OO design discussion, the goal for and application are
1. Ease of creation
2. Ease of maintenance
3. Ease of enhancement

## Coupling

Coupling is the degree to which one class knows about another class. If the only knowledge that class A has about class B, is what class B has exposed through its interface, then class A and class B are said to be loosely coupled…that's a good thing. If, on the other hand ,class A relies on parts of class B that are not part of class B's interface, then the coupling between the classes is tighter…not a good thing. In other words, if A knows more than it should about the way in which B was implemented, then A and B are tightly coupled.

Using this second scenario, imagine what happens when class B is enhanced. It's quite possible that the developer enhancing class B has no knowledge of class A, why would she? Class B's developer ought to feel that any enhancements that don't break the class's interface should be safe, so she might change some non interface
part of the class, which then causes class A to break.

## Cohesion

The term cohesion is used to indicate the degree to which a class has a single, well-focused purpose. Keep in mind that cohesion is a subjective concept. The more focused a class is, the higher its cohesiveness—a good thing. The key benefit of high cohesion is that such classes are typically much easier to maintain (and less frequently changed) than classes with low cohesion. Another benefit of high cohesion is that classes with a well-focused purpose tend to be more reusable than other classes. Instead of one class that does everything, we've broken the system into four main classes, each with a very specific, or cohesive, role. Because we've built these specialized, reusable classes, it'll be much easier to write a new report, since we've already got the database connection class, the printing class, and the file saver class, and that means they can be reused by other classes that might want to print a report

## Variables

■ Instance variables and objects live on the heap.
■ Local variables live on the stack.

## Passing Variables into Methods

## Methods can be declared to take primitives and/or object references

## The Java Spec says that everything in Java is pass-by-value. There is no such thing as "pass-by-reference" in Java.

Example:

```
package com.practice.basic;

/**
 *
 * @author rkumar
 * class to demo pass by Value and pass by reference
 *
 */
class Dog {
	private String name;
	public Dog(String name) {
		this.name = name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getName() {
		return name;
	}
}
public class PassbyValue {

	public static void foo(Dog someDog) {
		someDog.setName("Max");
		someDog = new Dog("Fifi");
		someDog.setName("RowFi");
	}

	public static void main(String[] args) {
		Dog d = new Dog("Rover");
		foo(d);
		System.out.println("dog name : "+d.getName());
	}
}
```

Pasted from <https://stackoverflow.com/questions/40480/is-java-pass-by-reference-or-pass-by-value>

The key to understanding this is that something like
`Dog myDog;`
is *not* a Dog; it's actually a *pointer* to a Dog.

What that means, is when you have

```
Dog myDog = new Dog("Rover");
foo(myDog);
```

you're essentially passing the *address* of the created `Dog` object to the `foo` method.
(I say essentially because Java pointers aren't direct addresses, but it's easiest to think of them that way)
Suppose the `Dog` object resides at memory address 42. This means we pass 42 to the method.
if the Method were defined as

```
public void foo(Dog someDog) {
  someDog.setName("Max");    // AAA
  someDog = new Dog("Fifi");  // BBB
  someDog.setName("Rowlf");  // CCC
}
```

let's look at what's happening.

- the parameter `someDog` is set to the value 42
- at line "AAA"
  - `someDog` is followed to the `Dog` it points to (the `Dog` object at address 42)
  - that `Dog` (the one at address 42) is asked to change his name to Max
- at line "BBB"
  - a new `Dog` is created. Let's say he's at address 74
  - we assign the parameter `someDog` to 74
- at line "CCC"
  - someDog is followed to the `Dog` it points to (the `Dog` object at address 74)
  - that `Dog` (the one at address 74) is asked to change his name to Rowlf
- then, we return

Now let's think about what happens outside the method:
Did `myDog` change?
There's the key.
Keeping in mind that `myDog` is a *pointer*, and not an actual `Dog`, the answer is NO. `myDog` still has the value 42; it's still pointing to the original `Dog` (but note that because of line "AAA", its name is now "Max" - still the same Dog; `myDog`'s value has not changed.)
It's perfectly valid to *follow* an address and change what's at the end of it; that does not change the variable, however.
Java works exactly like C. You can assign a pointer, pass the pointer to a method, follow the pointer in the method and change the data that was pointed to. However, you cannot change where that pointer points.
In C++, Ada, Pascal and other languages that support pass-by-reference, you can actually change the variable that was passed.
If Java had pass-by-reference semantics, the `foo` method we defined above would have changed where `myDog` was pointing when it assigned `someDog` on line BBB.
Think of reference parameters as being aliases for the variable passed in. When that alias is assigned, so is the variable that was passed in.

## Garbage Collector:

When Does the garbage Collector Run?
The garbage collector is under the control of the JVM. The JVM decides when to run the garbage collector. From within your Java program you can ask the JVM to run the garbage collector, but there are no guarantees, under any circumstances, that the JVM will comply. Left to its own devices, the JVM will typically run the garbage
collector when it senses that memory is running low. Experience indicates that when your Java program

makes a request for garbage collection, the JVM will usually grant your request in short order, but there are no guarantees. Just when you think you can count on it, the JVM will decide to ignore your request.

## an object is eligible for garbage collection when no live thread can access it.

## Garbage collector uses a mark and sweep algorithm, and for any given Java implementation that might be true, but the Java specification doesn't guarantee any particular implementation. You might hear that the garbage collector uses reference counting; once again maybe yes maybe no.

## Writing Code That Explicitly Makes Objects Eligible for Collection

An object becomes eligible for garbage collection when there are no more reachable references to it. Obviously, if there are no reachable references, it doesn't matter what happens to the object. For our purposes it is just floating in space, unused, inaccessible, and no longer needed.

1. Nulling a Reference
   first way to remove a reference to an object is to set the reference variable that refers to the object to null

2. Reassigning a Reference Variable
   We can also decouple a reference variable from an object by setting the reference variable to refer to another object.

3. Object Created inside the method
   Objects that are created in a method also need to be considered. When a method is invoked, any local variables created exist only for the duration of the method. Once the method has returned, the objects created in the method are eligible for garbage collection. There is an obvious exception, however. If an object is returned from the method, its reference might be assigned to a reference variable in the
   method that called it; hence, it will not be eligible for collection.
4. Isolating a Reference
   There is another way in which objects can become eligible for garbage collection, even if they still have valid references! We call this scenario "islands of isolation." A simple example is a class that has an instance variable that is a reference variable to another instance of the same class. Now imagine that two such instances exist and that they refer to each other. If all other references to these two objects are removed, then even though each object still has a valid reference, there will be no way for any live thread to access either object. When the garbage collector runs, it can usually discover any such islands of objects and remove them. As you can imagine, such islands can become quite large, theoretically containing hundreds of objects.

Example:

```
package com.practice.basic;

import java.util.Date;

/**
 *
 * @author rkumar
 * Demo implementation of garbage collection scenario's
 *
 */
```

```
public class GarbageCollectionDemo {

        GarbageCollectionDemo garbageCollectionDemo;

        public static Date getDate() {
                Date d = new Date();
                StringBuffer sb = new StringBuffer(d.toString());
                // Eligible for garbage collector if methods poped up from stack.
                System.out.println(sb);
                return d;
        }
        /**
         *
         * @param args
         * Function to show differnt kind of garbage collection eligible cases.
         */
        public static void main(String[] args) {
                // Nulling the Refernce

                StringBuffer sb = new StringBuffer("Hello");
                // Not eligible for garbage collection
                System.out.println("Sb : "+sb);
                sb = null;
                // Now eligible for garbage collection
                System.out.println("sb after: "+sb);

                // Reassigning a Reference Variable

                StringBuffer sb1 = new StringBuffer("Hello");
                StringBuffer sb2 = new StringBuffer("Halo");
            // Not eligible for garbage collector
                System.out.println(sb1);
                sb1 = sb2;
                // Sb1 is eligible for garbage collector

      // Object Created in inside method
                Date d = getDate();
                System.out.println(d);

                // Isolating a Reference
                GarbageCollectionDemo gb1 = new GarbageCollectionDemo();
                GarbageCollectionDemo gb2 = new GarbageCollectionDemo();
                GarbageCollectionDemo gb3 = new GarbageCollectionDemo();

                gb1.garbageCollectionDemo = gb2;
                gb2.garbageCollectionDemo = gb3;
                gb3.garbageCollectionDemo = gb1;

                // gb1, gb2 and gb3 eligible for garbage collector

        }
}
```

## Forcing Garbage Collector:

It is possible only to suggest to the JVM that it perform garbage collection. However, there are no guarantees the JVM will actually remove all of the unused objects from memory (even if garbage collection is run).

The garbage collection routines that Java provides are members of the Runtime class. The Runtime class is a special class that has a single object (a Singleton) for each main program. The Runtime object provides a mechanism for communicating directly with the virtual machine. To get the Runtime instance, you can use the
method Runtime.getRuntime(), which returns the Singleton. Once you have the Singleton you can invoke the garbage collector using the gc() method. Alternatively, you can call the same method on the System class, which has static methods that can do the work of obtaining the Singleton for you. The simplest way to ask for garbage collection (remember—just a request) is
System.gc();
Theoretically, after calling System.gc(), you will have as much free memory as possible. We say theoretically because this routine does not always work that way. First, your JVM may not have implemented this routine; the language specification allows this routine to do nothing at all. Second, another thread might grab lots of memory right after you run the garbage collector.

Example:

```
package com.practice.basic;

import java.util.Date;

/**
 *
 * @author rkumar
 * Class to show demo how garbage collector we can force.
 *
 */
public class GarbageCollectorFreeDemo {
        public static void main(String [] args) {
                Runtime rt = Runtime.getRuntime();
                System.out.println("Total JVM Memory before : "+rt.totalMemory());
                System.out.println("Total free Memory before : "+rt.freeMemory());

                Date d = null;

                for(int i=0; i<10000; i++) {
                        d = new Date();
                        d = null;
                }

                System.out.println("Total free Memory before : "+rt.freeMemory());
                // Garbage collector function
                rt.gc();
                System.out.println("Total free Memory after : "+rt.freeMemory());


        }
}
```

## Cleaning Up Before Garbage Collection—the finalize() Method

Java provides you a mechanism to run some code just before your object is deleted by the garbage collector. This code is located in a method named finalize() that all classes inherit from class Object. On the surface this sounds like a great idea; maybe your object opened up some resources, and you'd like to close them before your object is deleted. The problem is that, as you may have gathered by now, you can't count on the garbage collector to ever delete an object. So, any code that you put into your class's overridden finalize() method is not guaranteed to run. The finalize() method for any given object might run, but you can't count on it, so don't put any essential code into your finalize() method. In fact, we recommend that in general you don't override finalize() at all.

Tricky Little finalize() Gotcha's There are a couple of concepts concerning finalize() that you need to remember.
■ For any given object, finalize() will be called only once (at most) by the garbage collector.
■ Calling finalize() can actually result in saving an object from deletion.

** **String, StringBuilder, StringBuffer**

The StringBuilder class to the API, to provide faster, nonsynchronized StringBuffer capability. The StringBuilder
class has exactly the same methods as the old StringBuffer class, but StringBuilder is faster because its methods aren't synchronized. Both classes give you String-like objects that handle some of the String class's shortcomings (like immutability).

## The String Class

## In java, each character in a string isa 16-bit Unicode character. In java, string are objects.

```
String  str = new String();
```

## String is immutable.

```
    String s1 = "Spring";
    String s2  = s1 + "summer ";
    s1.concat("fall ");
    s2.concat(s1);
    s1 += "winter";
    System.out.println(s1  +  " " + s2);
```

Output : spring winter spring summer

Objects and Reference Variables:

There are two reference variables s1 and s2. There were a total of eight String Objects created as follows:
"spring", "summer" (lost), "spring summer", "fall" (lost), "spring fall" (lost), "spring summer spring" (lost),
"winter" (lost), "spring winter" (At this point "spring" is lost). Only two of the eight String Objects are not lost in this process.

## Important Facts About String and Memory

One of the key goals of any good programming language is to make efficient use of memory. As applications grow, it's very common for String literals to occupy large amounts of a program's memory, and there is often a lot of redundancy within the universe of String literals for a program. To make Java more memory efficient, the JVM sets aside a special area of memory called the "String constant pool." When the compiler encounters a String literal, it checks the pool to see if an identical String already exists. If a match is found, the reference to the new literal is directed to the existing String, and no new String literal object is created. (The existing String simply has an additional reference.) Now we can start to see why making String objects immutable is such a good idea. If several reference variables refer to the same String without even knowing it, it would be very bad if any of them could change the String's value.

You might say, "Well that's all well and good, but what if someone overrides the String class functionality; couldn't that cause problems in the pool?" That's one of the main reasons that the String class is marked final. Nobody can override the behaviors of any of the String methods, so you can rest assured that the String objects you are counting on to be immutable will, in fact, be immutable.

```
String s = abc  // creates one String object and one reference variable
String s = new String("abc"); // creates two objects, and one reference variable
```

## Because we used the new keyword, java will create a new String Object in normal (nonpool) memory, and s will refer to it. In addition, the literal "abc" will be placed in the pool.

## Arrays have an attribute (not a method), called length. You may encounter questions in the exam that attempt to use the length() method on an array, or that attempt to use the length attribute on a String. Both cause compiler errors—for example,

```
String x = "test";
System.out.println(x.length); // compiler error
```

Or

```
String[] x = new String[3];
System.out.println( x;length() ); // compiler error
```

## The StringBuffer and StringBuilder classes

The java.lang.StringBuffer and java.lang.StringBuilder classes should be used whenyou have to make a lot of modifications to strings of characters. As we discussed inthe previous section, String objects are immutable, so if you choose to do a lot of manipulations with String objects, you will end up with a lot of abandoned String objects in the String pool. (Even in these days of gigabytes of RAM, it's not a good idea to waste precious memory on discarded String pool objects.) On the other hand, objects of type StringBuffer and StringBuilder can be modified over and over again without leaving behind a great effluence of discarded String objects.

## A common use for StringBuffers and StringBuilders is file I/O when large, ever-changing streams of input are being handled by the program. In these cases, large blocks of characters are handled as units, and StringBuffer objects are the ideal way to handle a block of data, pass it on, and then reuse the same memory to handle the next block of data.

## StringBuffer vs StringBuilder

The StringBuilder class was added in Java 5. It has exactly the same API as the StringBuffer class, except StringBuilder is not thread safe. In other words, its methods are not synchronized. Sun recommends

that you use StringBuilder instead of StringBuffer whenever possible because StringBuilder will run faster (and perhaps jump higher). So apart from synchronization, anything we say about StringBuilder's methods holds true for
StringBuffer's methods, and vice versa.

## StringBuffer

```
StringBuffer sb = new StringBuffer("abc");
Sb.append("def");
System.out.println("sb = " +sb); // output is  "sb = abcdef"
```

All of the StringBuffer methods we will discuss operate on the value of the StringBuffer object invoking the method. So a call to sb.append("def"); is actually appending "def" to itself (StringBuffer sb). In fact, these method calls can be chained to each other.

```
StringBuilder sb = new StringBuilder("abc");
Sb.append("def").reverse().insert(3, "---");
System.out.println(sb);
```

Notice that in each of the previous two examples, there was a single call to new, concordantly in each example we weren't creating any extra objects. Each example needed only a single StringXxx object to execute

***** File Navigation and I/O**
Java.io : -
BufferedReader, BufferedWriter, File, FileReader, FileWriter, PrintWriter, and Console.

## A general discussion of I/O could include topics such as file I/O, console I/O, thread I/O, high-performance I/O, byte-oriented I/O, character-oriented I/O, I/O filtering and wrapping, serialization

I/O classes

■ **File:**
The API says that the class File is "An abstract representation of file and directory pathnames." The File class isn't used to actually read or write data; it's used to work at a higher level, making new empty files, searching for
files, deleting files, making directories, and working with paths.

■ **FileReader**
This class is used to read character files. Its read() methods are fairly low-level, allowing you to read single characters, the whole stream of characters, or a fixed number of characters. FileReaders are usually wrapped
by higher-level objects such as BufferedReaders, which improve performance and provide more convenient ways to work with the data.

■ **BufferedReader**
This class is used to make lower-level Reader classes like FileReader more efficient and easier to use. Compared to FileReaders, BufferedReaders read relatively large chunks of data from a file at once, and keep this data in a buffer. When you ask for the next character or line of data, it is retrieved from the buffer, which minimizes the number of times that time-intensive, file read operations are performed. In addition,
BufferedReader provides more convenient methods such as readLine(), that allow you to get the next line of characters from a file.

■ **FileWriter**
This class is used to write to character files. Its write() methods allow you to write character(s) or Strings to a file. FileWriters are usually wrapped by higher-level Writer objects such as BufferedWriters or PrintWriters, which provide better performance and higher-level, more flexible methods to write data.

■ **BufferedWriter**
This class is used to make lower-level classes like FileWriters more efficient and easier to use. Compared to FileWriters, BufferedWriters write relatively large chunks of data to a file at once, minimizing the number of times that slow, file writing operations are performed. The BufferedWriter class also provides a newLine() method to create platform-specific line separators automatically.

■ **PrintWriter**
This class has been enhanced significantly in Java 5. Because of newly created methods and constructors (like building a PrintWriter with a File or a String), you might find that you can use PrintWriter in places where you previously needed a Writer to be wrapped with a FileWriter and/or a BufferedWriter. New methods like format(), printf(), and append() make PrintWriters very flexible and powerful.

■ **Console**
This new, Java 6 convenience class provides methods to read input from the console and write formatted output to the console.

## Stream classes are used to read and write bytes, and Readers and Writers are used to read and write characters. Since all of the fi le I/O on the exam is related to characters, if you see API class names containing the word "Stream", for instance DataOutputStream.

## File

```
package com.practice.io;

import java.io.File;
import java.io.IOException;

/**
 *
 * @author rkumar Class to create a file and validate if file exists
 *
 */
public class SimpleFileDemo {
	// test method
	public static void main(String[] args) {
		try {
			File file = new File("demo.txt"); // It's only a object means file
											// not yet created
			System.out.println(file.exists()); // look for a real file
			file.createNewFile(); // file created
			System.out.println(file.exists());
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
}
```

## Using FileWriter and FileReader

## FileWriter and FileReader used to read character stream.

```
package com.practice.io;

import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

/**
 *
 * @author rkumar
 * class to describe FileWriter and FileReader
 *
 */
public class FileWriterAndReaderDemo {
	// test method
	public static void main(String [] args) {
		char[] ch = new char[50];
		int size = 0;
		try {
			// Writing the file
			File file = new File("fileWriterAndReader.txt");
			FileWriter fileWriter = new FileWriter(file);
			fileWriter.write("Hello Ritesh Chauhan");
			fileWriter.flush();
			fileWriter.close();

			//Reading the file
			FileReader fr = new FileReader(file);
			size = fr.read(ch); // passing char array to read from file
			for(char c: ch) {
				System.out.print(c);
			}
			fr.close();
		} catch(IOException e) {
			e.printStackTrace();
		}

	}
}
```

## flush() and close() method

When you write data out to a stream, some amount of buffering will occur, and you never know for sure exactly when the last of the data will actually be sent. You might perform many write operations on a stream before closing it, and invoking the flush() method guarantees that the last of the data you thought you had already written actually gets out to the file. Whenever you're done using a file, either reading it or writing to it, you should invoke the close() method. When you are doing file I/O you're using expensive and limited operating system resources, and so when you're done, invoking close() will free up those resources.

## When we were reading data back in, we put it into a character array. It being an array and all, we had to declare its size beforehand, so we'd have been in trouble if we hadn't made it big enough! We could have read the
data in one character at a time, looking for the end of file after each read(), but that's pretty painful too.

Because of these limitations, we'll typically want to use higher-level I/O classes like BufferedWriter or BufferedReader in combination with FileWriter or FileReader.

## PrintWriter and BufferedReader

## Take special note of what the readLine() method returns. When there is no more data to read, readLine() returns a null—this is our signal to stop reading the file.

```
package com.practice.io;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

/**
 *
 * @author rkumar
 *
 */
public class FileWithPrintWriter {
	/**
	 *
	 * @param args
	 */
	public static void main(String [] args) {
		try {
			// Writng file
			File file = new File("printWriter.txt");
			FileWriter fw = new FileWriter(file);
			PrintWriter printWriter = new PrintWriter(fw);
			printWriter.println("How are you Ritesh");
			printWriter.println("I am gud");
			fw.flush();
			fw.close();

			// reading file
			FileReader fr = new FileReader(file);
			BufferedReader br = new BufferedReader(fr);
			String line;
			// readline move curser to next line
			while((line = br.readLine()) != null) {
```

```
                    System.out.println(line);
                }
                br.close();

            } catch (IOException e) {
                e.printStackTrace();
            }


        }
}
```

## Directory and File

Creating a directory is similar to creating a file. first we create a Directory (File) object, then we create an actual directory using the following mkdir() method.

```
package com.practice.io;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

/**
 *
 * @author rkumar
 * Directory Demo Class
 *
 */
public class DirectoryDemo {

        // test method
        public static void main(String [] args) {
            try {

                //Creating Directory
                File mydir = new File("mydir");
                boolean exists = mydir.mkdir();
                System.out.println(exists);

                // Writing a file
                File myFile = new File(mydir, "demo.txt");
//                  PrintWriter pr = new PrintWriter(myFile);
//                  pr.write("new stuff");
//                  pr.flush();
//                  pr.close();
                FileWriter fw = new FileWriter(myFile);
                BufferedWriter bw = new BufferedWriter(fw);
                bw.write("Ritesh Kumar");
```

```
			bw.flush();
			bw.close();

			// Reading a file
			FileReader fr = new FileReader(myFile);
			BufferedReader br = new BufferedReader(fr);
			String line;
			while((line = br.readLine()) != null) {
				System.out.println(line);
			}
			br.close();
		} catch(IOException e) {
			e.printStackTrace();
		}

	}
}
```

## Console Class

## The console is the physical device with a keyboard and a display (like your Mac or PC). If you're running Java SE 6 from the command line, you'll typically have access to a console object, to which you can get a reference by invoking System.console(). Keep in mind that it's possible for your Java program to be running in an environment that doesn't have access to a console object, so be sure that your invocation of System.console() actually returns a valid console reference and not null. The Console class makes it easy to accept input from the command line, both echoed and nonechoed (such as a password), and makes it easy to write formatted output to the command line. It's a handy way to write test engines for unit testing or if you want to support a simple but secure user interaction and you don't need a GUI. On the input side, the methods you'll have to understand are readLine and readPassword. The readLine method returns a string containing whatever the user keyed in—that's pretty intuitive. However, the readPassword method doesn't return a string: it returns a character array. Here's the reason for this: Once you've got the password, you can verify it and then absolutely remove it from memory. If a string was returned, it could exist in a pool somewhere in memory and perhaps some nefarious hacker could find it.

Example:
Need to explore more

## Serialization

Transient variables can't be serialized
For working with Serialization, We need to implement Serializable interface which is marker interface which don't have methods.
Serializable interface should be implement with class which need to be serialized.

```
package com.practice.io;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
```

```
/**
 *
 * @author rkumar
 * Basic Serialization Demo
 *
 */
class Cat implements Serializable {
	/**
	 *
	 */
	private static final long serialVersionUID = 1L;
	String color;
	void showColor() {
		System.out.println("color show");
	}
}
public class SerializationBasicDemo {
	public static void main(String args[]) {
		Cat c = new Cat();
		try {
			FileOutputStream fileOutputStream = new FileOutputStream("demo.ser");
			ObjectOutputStream outputStream = new ObjectOutputStream(fileOutputStream);
			outputStream.writeObject(c);
			outputStream.close();
		} catch(Exception e) {
			e.printStackTrace();
		}

		try {
			FileInputStream fileInputStream = new FileInputStream("demo.ser");
			ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream);
			c = (Cat) objectInputStream.readObject();
			c.showColor();
			objectInputStream.close();

		} catch(Exception e) {
			e.printStackTrace();
		}
	}
}
```

## Serialization in inheritance
If a superclass is Serializable, then according to normal Java interface rules, all subclasses of that class automatically implement Serializable implicitly. In other words, a subclass of a class marked Serializable passes the IS-A test for Serializable, and thus can be saved without having to explicitly mark the subclass as Serializable. You simply cannot tell whether a class is or is not Serializable UNLESS you can see the class inheritance tree to see if any other super classes implement Serializable. If the class does not explicitly extend any other class, and does not implement Serializable, then you know for CERTAIN that the class is not Serializable, because class Object does NOT implement Serializable.

If you are a serializable class, but your superclass is NOT serializable, then any instance variables you INHERIT from that superclass will be reset to the values they were given during the original construction of the object. This is because the non serializable class constructor WILL run!.

## If you serialize a collection or an array, every element must be serializable! A single non-serializable element will cause serialization to fail. Note also that while the collection interfaces are not serializable, the concrete collection classes in the Java API are.

## Serialization Is Not for Statics
Finally, you might notice that we've talked ONLY about instance variables, not static variables. Should static variables be saved as part of the object's state? Isn't the state of a static variable at the time an object was serialized important? Yes and no. It might be important, but it isn't part of the instance's state at all. Remember, you should think of static variables purely as CLASS variables. They have nothing to do with individual instances. But serialization applies only to OBJECTS. And what happens if you deserialize three different Dog instances, all of which were serialized at different times, and all of which were saved when the value of a static variable in class Dog was different. Which instance would "win"? Which instance's static value would be used to replace the one currently in the one and only Dog class that's currently loaded? See the problem? Static variables are NEVER saved as part of the object's state…because they do not belong to the object!

## Regex Pattern

\d A digit
\s A whitespace character
\w A word character