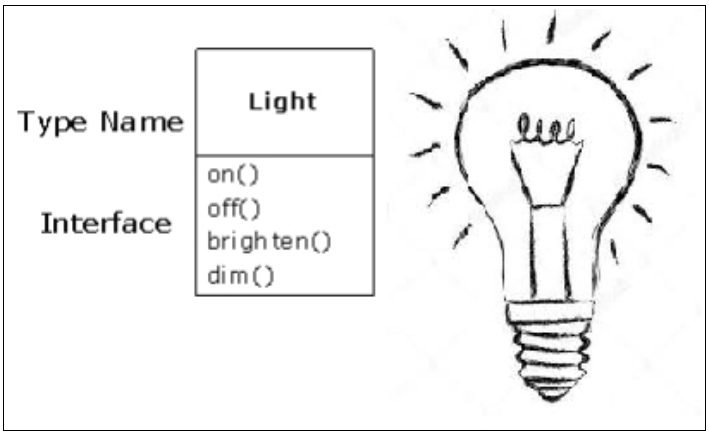
Table of Contents

Chapter 1 . Introduction to objects

- Class is nothing but a type, i.e. class shows its represents some type. Whenever an object is been created from a class they all are nothing but image of the same class. Hence its clear that class which is a type gives an frame work or design upon which object is created. This is called **ABSTRACTION**.

- **A Object has an Interface :**

Putting member variables in a class and access to them is given only through by member method which are also called as interfaces in layman term (not java proper, java interface is something different). This way of keeping member variables private and allowing access only through interfaces is called **ENCAPSULATION**.

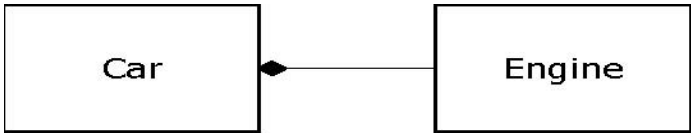



```
Light lt = new Light();
lt.on();
```

Above program we can see how encapsulation has been used.

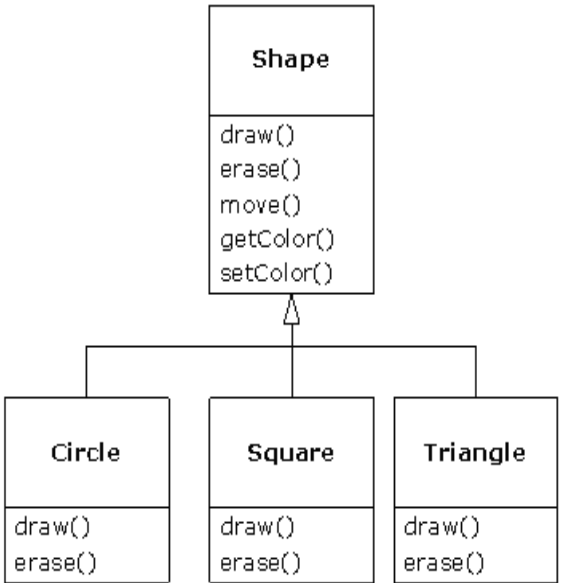
- Reusing the implementation :

Simplest way to reuse a class is by using a object of that class inside a new class. So a new class can be composed by creating many objects which is called **COMPOSITION**. If composition happens dynamically then its called **AGGREGATION**. Composition is often referred to as a “has-a” relationship, as in “A car has an engine”.



Diamond in UML diagram above shows composition.

- **Inheritance and Interchangable objects with polymorphism :**



We know that with inheritance we can have overridden methods too. Lets say Shape class have one more method doSomething() as shown below.

```
void doSomething(Shape shape) {
       shape.erase();
       // ...
       shape.draw();
    }
```

Above we can see method doSomething() speak to any shape. Whether its circle or square it can draw or erase. Now lets say some part of the program calls doSomething() as below

```
Circle circle = new Circle();
Triangle triangle = new Triangle();
Line line= new Line();
doSomething(circle);
doSomething(triangle);
 doSomething(line);
```

Above we can see the in doSomething(circle) circle(a sub-class object) has been passed to the method who was expecting a shape. This treating of sub-class object as super-class is called **UPCASTING.** (This happens because of polymorphism where subclass ref can be assigned to superclass).

Above when doSomething(circle) is called it gets into the body and found that shape.draw() or shape.erase() needs to be called but at compile time compiler will be not sure that which piece of code will be executed. In traditional programming execution flow was known during compile time which is called **EARLY BINDING** but this will produce falls results with java . So java uses **LATE BINDING** i.e. execution flow will be determined during Run Time.

- Example for the Late binding scenario is given below

```
class BaseClass
{
	int a=5,b=5;
	int sumInt()
	{
		int sum = 0;
		sum = a+b;
		return sum;
	}

	void printSum(BaseClass obj)
	{
		System.out.println("this is base class method "+obj.sumInt());
	}
}

class DerivedClass extends BaseClass
{
	int i=10,j=10;
	int sumInt()
	{
		int sum = 0;
		sum = i+j;
		return sum;
	}

}

public class PolymrphClass {

	public static void main(String[] args)
	 {
		// TODO Auto-generated method stub
		BaseClass bc = new BaseClass();
		DerivedClass dc = new DerivedClass();
		System.out.println(bc.sumInt());
		System.out.println(dc.sumInt());

		bc.printSum(dc);

		bc = dc;
		System.out.println(bc.sumInt());
	}

}
```

```
Output :
10
20
this is base class method 20
20
```

```java
package Chap1TIJ;

class BaseClass1
{
	String str1;
	BaseClass1(String str)
	{
		str1 = str;
	}
	void PrintBaseMet1(BaseClass1 obj)
	{
		System.out.println(str1);
	}
	void PrintBaseMet2(BaseClass1 obj)
	{
		System.out.println("overloaded base");
	}
	void PrintMet3()
	{
		System.out.println("overridden base");
	}
}

class DeriveClass1 extends BaseClass1
{
	String str2;
	DeriveClass1(String str)
	{
		super("derive base");
		str2 = str;
	}
	void PrintDeriveMet1(DeriveClass1 obj)
	{
		System.out.println(str2);
	}
	void PrintDeriveMet2(DeriveClass1 obj)
	{
		System.out.println("overloaded derive");
	}
	void PrintMet3()
	{
		System.out.println("overridden derive");
	}
}

public class CheckUpcastingThoroughly
{
	public static void main(String[] args)
	{
		BaseClass1 bc1 = new BaseClass1("Base");
		DeriveClass1 bc2 = new DeriveClass1("Derive");

		bc1.PrintBaseMet1(bc1);
		bc1.PrintBaseMet2(bc1);
		bc1.PrintMet3();

		bc2.PrintDeriveMet1(bc2);
```

```
            bc2.PrintDeriveMet2(bc2);
            bc2.PrintMet3();

            bc1.PrintBaseMet1(bc2);
            bc1.PrintBaseMet2(bc2);
            bc2.PrintMet3();
      }
}
Output:
Base
overloaded base
overridden base
Derive
overloaded derive
overridden derive
Base
overloaded base
overridden derive
```

➢ In above example we can see that when method has been overloaded or overridden then it will work accordingly and upcasting don't take place. Only in case of method which is unique to superclass and doesn't present in subclass like method PrintBaseMet1() upcasting take place.

➢ **<u>Containers and Parameterized type (Generics)</u> :**

- In general, one wont know how many objects he needs to be created and how to store them until during Run Time. Java always provides solution by creating more objects. The new type of objects that solves this particular problem holds references to other objects. This new objects are generally called as Container(or collection also).

- Collection stores only objects. So when references are stored in containers they upcast them to objects hence they lose their character and while fetching back we get objects. To get the particular object references we need to downcast this objects which is called **DOWNCASTING.**

- As during upcasting we go specific to the more general(derived to base) type so in downcasting its opposite, we go from general to specific. Upcasting is easy as we need circle is shape but downcasting may arise problem because a object can be shape or circle.

- However downcasting is not that dangerous because if you got to the wrong object then Error will occur called exception. But still downcasting and runtime checks required extra time for running program and extra effort from the programmer. Hence we use **PARAMETERIZED TYPE** mechanism.

- Parameterized type is a class which compiler customized directy to use with particular type. Example for parameterized container compiler customize the container so that it could accept only shapes and fetch only shapes.

- One big change in Java SE5 is addition of parameterized types called **GENERICS.** Generics are recognized with the angle brackets with types inside. Example a ArrayList that holds shapes can be created like this.

```
ArrayList<Shape> shapes = new ArrayList<Shape>();
```

Chapter 2 . Everything is a object

- A reference of a object is separate than the object. A reference can exist even if there is no object associated with it.

- Where storage live :

1. Registers : It is the fastest storage because registers are present in processor itself. But number of registers are very few and programmer don't have direct control of them.
2. Stack : This is the second fastest after register as processor have direct support for them through *stack pointers.* They are a part of RAM and Java System must know while creating the program, the exact lifetime of all the items that lives on the stack. This put constraint on flexibility of the program. So java storage exist on the stack particularly references but java objects themselves are not stored on it.
3. Heap: It is general purpose storage also part of the RAM where objects lives. For heap, java system don't need to know how long the storage must stay on the heap. This gives a good amount of flexibility. Whenever new keyword is used to create a object, the storage is allocated on the heap. But heap take more time to allocate and cleanup storage compare to stack.
4. Constant storage: Constant values are always placed in programmes but sometime they are placed in ROM in embedded systems.
5. Non-RAM storage: Sometime we need objects to stay outside the programme, even when program is not running, outside the control of programme. Like *streamed objects* which are turn into stream of bytes to send from one machine to other machine. Also *persistence objects* which are stored on the disk so that they can hold their state even when program is terminated.

- Static keyword :

- When one want to use a single storage for a field no matter how many objects holds that field they still share the same value. Also when one want a method which is not associated with any object then one should do that using static keyword.

- Example

```
class StaticTest {
static int i = 47;
```

When one will create two objects as shown below

```
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
```

Both st1.i and st2.i will have the same value 47. If the value get changed through one object then the same value will reflect for the other.

- One can even use static fields directly using class name. Example for above we can use field 'i' directly as StaticTest.i.
- Static methods too can be called in the same way with class name. Lets say there is a static method increment() in class StaticTest then one can call it directly without creating it object as StaticTest.increment().
- But it is also true that one can call static method using an object too.
- (KnS) Static methods can't be overridden as they belongs to class.
- (KnS) Only methods, variables and top-level nested classes can be marked as static.
- (KnS) Constructors, classes, Interfaces, Inner classes, Inner class methods and instance variables, Local variables all these can't be marked as static.
- Non-static methods can't be called through static methods even they are a part of same class. Example

```
package ch2EverythingIsAnObject;

class StaticAccess

{
	int a;
	static int b;

	static void method1()
	{
	  //a=10;//cannot access non-static variables
	  b=20;
	  //method2(); //cannot call a non-static method
	   System.out.println("inside static method");
	   System.out.println("value of b :"+b);

	}

	void method2()
	{
	  a=30;
	  b=40;//non-static method can access static variable
	   System.out.println("inside non-static method");
	   System.out.println("value of a:"+a+" "+"value of a:"+b);
	   method1();//non-static method can call static method
	}
}

public class StaticAccessingStatic
{
	public static void main(String args[])
	{
		StaticAccess sa = new StaticAccess();
		sa.method1();
		sa.method2();
```

```
            }
        }
        Output:
        inside static method
        value of b :20
        inside non-static method
        value of a:30 value of a:40
        inside static method
        value of b :20
```

Above one can see method2() cant be called from a static method but reverse is possible. (Note: see also chapter 5 for why keyword 'this' cant be used in static method)

```
package ch2EverythingIsAnObject;

class Static2
{
	static int i1, i2;
	int i3;
	static void increment()
	{
		i1++;
		i2++;

	}
	static void decrement()
	{
		i1--;
		i2--;
	}
	static void printValue()
	{
		System.out.println(i1+" "+i2);
	}
	void sum()
	{
		int sum = 0;
		sum = i1+i2;
		printValue();//a non-static member can call a static
		System.out.println("sum : "+sum);

	}
	static void AccessNonStatic()
	{
		//a static method cannot call a non-static members
	    // i3 = 0;
		//System.out.println(sum());
	}
}
public class StaticMembers {
	public static void main(String[] args) {
		// calling members by classname
		System.out.println("calling with classname");
		System.out.println(Static2.i1+" "+Static2.i2);
		Static2.increment();
		Static2.printValue();
		Static2.decrement();
		Static2.printValue();
```

```
            //Static2.sum();//cant call non-static member using classname

            System.out.println("calling with object");
            Static2 stc = new Static2();
            System.out.println(stc.i1+" "+stc.i2);
            stc.increment();
            stc.printValue();
            stc.decrement();
            stc.printValue();

            System.out.println("calling non-static members");
            stc.increment();
            stc.sum();
            }
}
Output:
calling with classname
0 0
1 1
0 0
calling with object
0 0
1 1
0 0
calling non-static members
1 1
sum : 2
```

- **Special case : Primitive types :**

Instance of creating variables by using new, which place them on the heap, java uses a alternative where variables are automatically created without any references. This new types are called primitive types. Primitive types holds the values directly and they are placed on the stack hence they are much more efficient.

- Wrapper class allows one to make non-primitive objects who represents that primitive types.

Ex. `char c = 'x';`

```
Character ch = new Character(c);
```

Or one could also use:

```
Character ch = new Character('x');
```

Java SE5 *autoboxing* will automatically convert from a primitive to a wrapper type:

```
Character ch = 'x';
```

and back:

```
char c = ch;
```

- **High Precision numbers :**

Java provides two classes to perform high-precision arithmetic: BigInteger and BigDecimal. Although they fits as the same category of wrapper classes but either don't have primitive analogue. One can do everything that one does with int and char but one must use method calls instead of operators. Here the operation is bit slow because big sizes are involved.

- **Scoping :**

1. Scope is java is determined by curly braces {}.

Ex.

```
{
int x = 12;
// Only x available
{
int q = 96;
// Both x & q available
}
// Only x available
// q is "out of scope" :
}
```

2. Scope of Objects:

Lifetime of objects is not same as that of primitives.

```
{
String s = new String("a string");
} // End of scope
```

Even as above scope of reference 's' is not longer exist but objects still lives on the heap which is removed by garbage collection when its find that no reference is referencing to it.

- The argument list :

```
class Return1
{
	int a = 10, b = 10;
	int sum = 0;
	int retSum()
	{
		sum = a+b;
		return sum;
		//System.out.println("after return");//this wont execute
	}
	void retNothing()
	{
		System.out.println(sum);
		return;//its not compulsory but still code runs fine
```

```
        }
}
```

'return' keyword return the data from a method. If we write any code after the return method then there will be compiler error as shown above in method **retSum()**. Also when you don't want to return anything one can mention **void** as return type. Also as seen above in method **retNothing()** you can still use return keyword (itss not mandatory)and it will work fine.

The static keyword can be used in **3** scenarios

1. static variables
2. static methods
3. static blocks of code.

Static Variables :

- It is a variable which **belongs to the class** and **not** to **object**(instance)
- Static variables are **initialized only once** , at the start of the execution . These variables will be initialized first, before the initialization of any instance variables
- A **single copy** to be shared by all instances of the class
- A static variable can be **accessed directly** by the **class name** and doesn't need any object
- Syntax : <***class-name>.<variable-name>***

Static Method :

- It is a method which **belongs to the class** and **not** to the **object**(instance)
- A static method **can access only static data**. It can not access non-static data (instance variables)
- A static method **can call only** other **static methods** and can not call a non-static method from it.
- A static method can be **accessed directly** by the **class name** and doesn't need any object
- Syntax : <***class-name>.<method-name>***
- A static method cannot refer to "this" or "super" keywords in anyway

Side Note:

- main method is static , since it must be be accessible for an application to run , before any instantiation takes place.

Static Block :

The static block, is a block of statement inside a Java class that will be executed when a class is first loaded in to the JVM

```
1  class Test{
2   static {
3   //Code goes here
4   }
5  }
```

A **static block helps to initialize the static data members**, just like constructors help to initialize instance members.

Chapter 3 : Operators

- Assignment :

```
//: operators/Assignment.java
// Assignment with objects is a bit tricky.
import static net.mindview.util.Print.*;
class Tank {
int level;
}
public class Assignment {
public static void main(String[] args) {
Tank t1 = new Tank();
Tank t2 = new Tank();
t1.level = 9;
t2.level = 47;
print("1: t1.level: " + t1.level +
", t2.level: " + t2.level);
t1 = t2;
print("2: t1.level: " + t1.level +
", t2.level: " + t2.level);
t1.level = 27;
print("3: t1.level: " + t1.level +
", t2.level: " + t2.level);
}
} /* Output:
1: t1.level: 9, t2.level: 47
2: t1.level: 47, t2.level: 47
3: t1.level: 27, t2.level: 27 */
```

Note: import static net.mindview.util.Print.* above provide direct use of print.

Above we can see that during 3rd output both t1.level and t2.level become 27. This happens because **reference** t2 is assigned to t1. This is called **ALIASING**. To avoid the aliasing use t1.level = t2.level.

- Aliasing during method call:

```
package ch3Operators;

class Letter
{
	char c;
}
public class Ch3Aliasing {
	static void f(Letter y)
	{
		y.c='z';
	}
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Letter x = new Letter();
		x.c='a';
		System.out.println("1:x.c: "+x.c);
		f(x);
		System.out.println("1:x.c: "+x.c);
		}
}
```

```
//Output:
// 1:x.c: a
 // 1:x.c: z
```

Chapter 4 : Controlling Execution

- **Iteration :**

Example :

```
//: control/WhileTest.java
// Demonstrates the while loop.
public class WhileTest {
      static boolean condition()
      {
      boolean result = Math.random() < 0.8;
      System.out.print(result + ", ");
      return result;
      }
      public static void main(String[] args)
      {
      while(condition())
      System.out.println("Inside 'while'");
      System.out.println("Exited 'while'");
      }
}
```

Above math.random() generate a number between(0 to 1) . The Boolean value get stored in result which is used as a condition in while. 'Indisde while' will be executed till the result is found true otherwise output will be 'Exited while'.

- **For :**

Syntax for 'for' loop is given as

```
for(initialization; Boolean-expression; step)
        statement
```

Above we can put more than one statement separated by comma in place of initialization and step. Example for the same is show below.

```
//: control/CommaOperator.java
public class CommaOperator {
      public static void main(String[] args)
      {
            for(int i = 1, j = i + 10; i < 5; i++, j = i * 2)
            {
            System.out.println("i = " + i + " j = " + j);
            }
      }
}
Output:
i = 1 j = 11
i = 2 j = 4
i = 3 j = 6
i = 4 j = 8
```

- **<u>Foreach Syntax</u> :**

It is useful way to use 'for'i.e. by using foreach one can easily iterate through the arrays. Example is shown below.

```
//: control/ForEachFloat.java
import java.util.*;
public class ForEachFloat {
public static void main(String[] args) {
Random rand = new Random(47);
float f[] = new float[10];
for(int i = 0; i < 10; i++)
f[i] = rand.nextFloat();
for(float x : f)//remember foreach never ends with semicolon
System.out.println(x);
}
} /* Output:
0.72711575
0.39982635
0.5309454
0.0534122
0.16020656
0.57799757
0.18847865
0.4170137
0.51660204
0.73734957
```

Foreach can be used for a method that returns a array. As in string the method toCharArray() returns an array of characters so that one can easily iterate through characters in the string.

Example:

```
//: control/ForEachString.java
public class ForEachString {
      public static void main(String[] args) {
      for(char c : "An African Swallow".toCharArray() )
      System.out.print(c + " ");
      }
}
Output:
A n A f r i c a n S w a l l o w
```

It should be also remember that foreach never **ends** with a semicolon(;).

- There are some keywords which provide *unconditional branching* i.e. branch happens without any test. These include **return, break** and **continue.** Break quits the loop without executing the remaining statements and continue quits the current iteration and goes to the beginning of the loop to execute the next iteration.

- Goto cant be used in java. But label can be used with break and continue. As shown in the following program

```
//: control/LabeledFor.java
// For loops with "labeled break" and "labeled continue."
import static net.mindview.util.Print.*;
public class LabeledFor {
      public static void main(String[] args) {
      int i = 0;
      outer: // Can’t have statements here
      for(; true ;) { // infinite loop
            inner: // Can’t have statements here
            for(; i < 10; i++) {
            print("i = " + i);
            if(i == 2) {
            print("continue");
            continue;
      }
      if(i == 3) {
      print("break");
      i++; // Otherwise i never
      // gets incremented.
      break;
      }
      if(i == 7) {
      print("continue outer");
      i++; // Otherwise i never
      // gets incremented.
      Continue outer;
      }
      if(i == 8) {
      print("break outer");
      break outer;
      }
      for(int k = 0; k < 5; k++) {
      if(k == 3) {
      print("continue inner");
      continue inner;
      }
      }
      }
      }
      // Can’t break or continue to labels here
      } } /* Output:
      i = 0
      continue inner
      i = 1
      continue inner
      i = 2
      continue
      i = 3
      break
      i = 4
      continue inner
      i = 5
      continue inner
      i = 6
      continue inner
      i = 7
      continue outer
      i = 8
      break outer
```

Chapter 5 : Initialization and cleapup

➔ Constructors don't have return type.

➔ If one is using overloaded methods and if the methods look like sum(int i , int j) and if you have two values say byte a and byte b and if u send sum (a, b) it will still execute because byte will be promoted to int. But the reverse is not possible i.e. if method arguments are byte and if one is sending int then there will be error but one can cast and send. example sum((int)a, (int)b). In general if casting rule applied to method parameters.

➔ One cant use different return value types for overloading methods. Only argument list differs.

➔ While creating a class one don't need to write default constructor because compiler creates that but when one is writing a constructor himself then compiler don't create a default constructor so the programmer has to mention default constructor himself if he wants to create such object. But if he is not using default constructor but only parameterized one then it can be ignored.

```
package ch5InitializationNCleanUp;

class DefConst1
{
	int a;

	DefConst1(int i)
	{
		a=i;
	}
	void printval()
	{
		System.out.println("value of a : "+a);
	}
}

public class Ch5DefaultConstructorNeeded {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		DefConst1 dc = new DefConst1(1);
		dc.printval();
		//DefConst1 dc1 = new DefConst1();
		//Error 'the constructor DefConst1() is undefined
	}
}
Output:
value of a : 1
```

➔ To distinguish overloaded methods we use different argument list to differentiate them. But one can even change the argument order for differentiation. Example sum (int a , byte b) is different than sum(byte a , int b).

- **<u>The 'this' keyword</u> :**

➔ 'this' keyword can be used only in non-static methods.

➔ It is used when one needs a reference for the current object.

➔ Example

```
//: initialization/PassingThis.java
class Person {
public void eat(Apple apple) {
Apple peeled = apple.getPeeled();
System.out.println("Yummy");
}
}
class Peeler {
static Apple peel(Apple apple) {
// ... remove peel
return apple; // Peeled
}
}
class Apple {
Apple getPeeled() { return Peeler.peel(this); }
}
public class PassingThis {
public static void main(String[] args) {
new Person().eat(new Apple());
}
} /* Output:
Yummy
*///:~
```

➔ 'this' can be returned using return this.

➔ <u>Calling constructor from constructor</u> :

Sometime when one write many constructors then he can call one constructor from another just to avoid duplication of code. Also we know 'this' means 'this object' or 'current object' when used in a method as shown in above example. But when 'this' is used in a constructor it takes different meaning. Example is shown below.

```
//: initialization/Flower.java
// Calling constructors with "this"
import static net.mindview.util.Print.*;
public class Flower {
int petalCount = 0;
String s = "initial value";
Flower(int petals) {
petalCount = petals;
print("Constructor w/ int arg only, petalCount= "
+ petalCount);
}
```

```
Flower(String ss) {
print("Constructor w/ String arg only, s = " + ss);
s = ss;
}
Flower(String s, int petals) {
this(petals);
//! this(s); // Can't call two!
this.s = s; // Another use of "this"
print("String & int args");
}
Flower() {
this("hi", 47);
print("default constructor (no args)");
}
void printPetalCount() {
//! this(11); // Not inside non-constructor!
print("petalCount = " + petalCount + " s = "+ s);
}
public static void main(String[] args) {
Flower x = new Flower();
x.printPetalCount();
}
} /* Output:
Constructor w/ int arg only, petalCount= 47
String & int args
default constructor (no args)
petalCount = 47 s = hi
*///:~
```

➔ Above example one should note that 'this' can't be called twice one after another for a constructor or two constructor can't be called compiler will point to the first 'this' and will say that constructor call should be the first statement. Also one can't call constructor from a method.

➔ Also one should keep in mind that 'this' keyword for constructors could be used only inside a same class. For subclass we used keyword 'super' to call superclass constructor.

```
package ch5InitializationNCleanUp;

class ThisNSuperConstructor1
{
	String str="string";
	ThisNSuperConstructor1(int i)
	{
		System.out.println("Inside ThisNSuperConstructor1 "+i);
		//ThisNSuperConstructor1(str);
	}
	ThisNSuperConstructor1(String s)
	{
		this(20);//You can use 'this' only within the same class
	}
}

class ThisNSuperConstructor2 extends ThisNSuperConstructor1
```

```
{
	ThisNSuperConstructor2(int j)
	{
		super(10);//superclass constructor could be only called through 'super'
		this.str = ("string");
		System.out.println("Inside ThisNSuperConstructor2 "+j);
	}
}

public class ThisNSuperConstructor {

	public static void main(String[] args)
	{
		ThisNSuperConstructor2 sc2 = new ThisNSuperConstructor2(4);
	}
}
Output :
Inside ThisNSuperConstructor1 10
Inside ThisNSuperConstructor2 4
```

➔ Private Constructors :

If a constructor of a class marked private then subclass cannot extend it in a generic way. To extend such a superclass one needs to define another constructor apart from private constructor which can be called through subclass.

```
package ch7ReusingClasses;

class Fruit
{
	String fruitColor;
	private Fruit()
	{
		//Apple cannot extends fruit if only private constructor
		//is present
	}

	Fruit(String color)
	{//Only if non-private constructor is present then it can be extended
		System.out.println("fruit is "+color);
	}
}

class Apple extends Fruit
{
	Apple()
	{
		super("red");
	}
}

public class PrivateConstructorClass
{

	public static void main(String[] args)
	{
		Apple apple = new Apple();
```

```
	}
}
Output:
fruit is red
```

➔ We know that we can't use 'this' inside a static method. Because we know one cannot call a non-static methods from a static one even they are in the same class and keyword 'this' will give reference of the current object which enable static method to call non-static members which should be avoided.

```
class AClass
{
	void printMsg()
	{
		System.out.println("Inside non static method");
	}
}

class BClass
{
	static void methodstc(AClass ac)
	{
		System.out.println("Inside static method");
		ac.printMsg();
		/*AClass ac1 = new AClass();
		ac1.printMsg();*///This is legal. one can create an object and call f
		rom static
		//this.method2();//cant use this inside static method
		//method2();//cant make a static reference to the non-static method
	}
	static void method1()
	{
		System.out.println("Simple static method");
			//methodstc(new AClass());//static meth can call another //static
			method
	}

	void method2()
	{
		System.out.println("Inside method");
		method1();//non-static methods can call static one.
	}
}

public class ThisStatic {

	public static void main(String[] args) {
		AClass acc = new AClass();
		BClass bc = new BClass();
		bc.methodstc(acc);
		bc.method2();
	}
}
Output:
Inside static method
Inside non-static method
Inside method
Simple static method
```

➔ In the above example one can see that in a static method 'methodstc()' one can create an object and call a non-static method from a static one. I.e. if one gets a reference to an object inside static method either by creating an object or as a method parameter inside an static method then non-static method can be called using this reference.

➢ **Cleanup : Finalization and garbage collection :**

➔ As we know when one do '**new**' for a class an object got created . This object occupies a memory on heap. When it is not referred by any reference then it will be garbage collected. Sometime the memory allocation it done through other medium then **new.** In that case we need to use finalize() then garbage collector first call finalize method and then will do normal garbage collection. One can mention code to remove that specially allotted memory object in finalize().

➔ But how one can allot memory to some object without having **new.** As everything in java is an object. It happen because of native methods. Languages like C or C++ does that. So now we know that when to use finalize().

➔ Example :

```
class FClass
{
	boolean CheckOut = false;
	FClass(boolean checkout)
	{
		CheckOut = checkout;
	}
	void checkIn()
	{
		CheckOut = false;
	}
	protected void finalize()
	{
		if(CheckOut)
		{
			System.out.println("inside finalize");
		}
	}
}

public class FinalizeClass {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		FClass fc = new FClass(true);
		fc.checkIn();
		new FClass(true);
		System.gc();
	}
}
```

```
Output : inside finalize
```

The termination condition is that all **Book** objects are supposed to be checked in before they are garbage collected, but in **main( )**, a programmer error doesn't check in one of the books. Without **finalize( )** to verify the termination condition, this can be a difficult bug to find.

- **How Garbage Collector works :**

➔ Most common technique for GC is reference counting. What is done here is a reference counter will be given to each object. This reference counter will indicate how many references are referencing to particular object. While JVM does garbage collector it goes through all the objects. When it found that reference counter for any object is 'zero' then that object will get collected. But its an slow technique.

➔ Other fast GC technique JVMs use is Adaptive garbage collector. To do that there are different variants. One of them is Stop-Copy. What is done here is JVM stops the program for GC and go through all the references present on stack and static memory. It will trace all the objects they point to and it also find references present in those objects and also trace those object which they point to. Then it will just copy all the live objects into another part of the heap and remaining objects will be garbage collected.

➔ The disadvantage of stop-copy(also called as copy collector) is that it creates two separate memory locations. Second when program becomes stable and generate no garbage still copy collect creates two memory location which is wasteful and to deal with this GC have another method (here where adaptive part of it come into play) which is called as mark-sweep.

➔ What mark-sweep does is jvm will go through all the references in stack and static memory and trace all the live objects accordingly. When it finds any live object then it will mark it with a flag but no sweep followed yet. Only when all the marking process is completed then the sweep occurs. During the sweep the dead objects are released.

➔ There are other possible speedup possible in JVM like just in time(JIT) compiler. Here this compiler will partially or fully compile the program in native machine code so that it doesn't need to be interpreted by JVM and thus runs faster.

- **Order of Initialization :**

➔ When one creates an object by calling **new()** on it then its constructor will be called but before that all the variables including construction of object variables. Example

```
class OrderOfInitialztn1
{
      int i;
      OrderOfInitialztn1(int a)
```

```
	{
		i = a;

		System.out.println("object"+a+" will have "+a+" value");
	}
}
class OrderOfInitialztn2
{
	int j;
	OrderOfInitialztn1 oi1 = new OrderOfInitialztn1(1);
	OrderOfInitialztn2()
	{
		System.out.println("value of j variable is "+j);
		j = 10;
		System.out.println("value of j variable is "+j);
		System.out.println("inside OrderOfInitialztn2 class");
		oi3 = new OrderOfInitialztn1(33);
	}
	void OrderOfInitFn()
	{
	System.out.println("inside the function");
	}
	OrderOfInitialztn1 oi2 = new OrderOfInitialztn1(2);
	OrderOfInitialztn1 oi3 = new OrderOfInitialztn1(3);
}
public class OrderOfInitialization {
	public static void main(String[] args) {
		OrderOfInitialztn2 ooi1 = new OrderOfInitialztn2();
		ooi1.OrderOfInitFn();
	}
}
Output :
object1 will have 1 value
object2 will have 2 value
object3 will have 3 value
value of j variable is 0
value of j variable is 10
inside OrderOfInitialztn2 class
object33 will have 33 value
inside the function
```

➔ When objects have been created using 'static' reference variables then scenario will be different as shown in the following example.

```
class StaticInitializatn1
{
	static int a;
	StaticInitializatn1(int i)
	{
		a = i;
		System.out.println("value is "+a);
	}
	void SIFunctn()
	{
		System.out.println("method("+a+")");
	}
}
```

```
class StaticInitializatn2
{
	static StaticInitializatn1 st1 = new StaticInitializatn1(1);
	StaticInitializatn2()
	{
		st2.SIFunctn();
	}
	static StaticInitializatn1 st2 = new StaticInitializatn1(2);

}
class StaticInitializatn3
{
	StaticInitializatn1 st3 = new StaticInitializatn1(3);
	static StaticInitializatn1 st4 = new StaticInitializatn1(4);
	StaticInitializatn3()
	{
		st4.SIFunctn();
	}
	static StaticInitializatn1 st5 = new StaticInitializatn1(5);

}
public class StaticInitializatn {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		StaticInitializatn2 stc2 = new StaticInitializatn2();
		StaticInitializatn3 stc3 = new StaticInitializatn3();
		StaticInitializatn2 stc4 = new StaticInitializatn2();
	}
}
Output :
value is 1
value is 2
method(2)
value is 4
value is 5
value is 3
method(3)
method(3)//method(3) because 'a' is static
```

In the above example we can see how the execution flow is going. The order of initialization is statics first, if they haven't been initialized already by previous object creation i.e. static initialization is executed only **once** always, after that the non-static one. Also we can see object creation with static reference is first executed. After that object creation with non-static reference(in example above st3) then constructors is executed.

➔ Static members of a class can be put inside a 'static block' which will be executed once. Example is shown below.

```
class StaticBlock1
{
	StaticBlock1(int i)
	{
		System.out.println("StaticBlock1("+i+")");
	}
	void fn(int a)
```

```
	{
		System.out.println("fn("+a+")");
	}
}

class StaticBlock2
{
	static StaticBlock1 st1;
	static StaticBlock1 st2;
	static
	{
		st1 = new StaticBlock1(1);
		st2 = new StaticBlock1(2);
	}
	StaticBlock2()
	{
		System.out.println("StaticBlock2");
	}
}

public class StaticBlock {
	public static void main(String[] args)
	{
		System.out.println("main method");
		StaticBlock2.st1.fn(1);//an static member can be accessed through class
					name.
	}

}
Output :
main method
StaticBlock1(1)
StaticBlock1(2)
StaticBlock2
fn(1)
```

➔ Similarly we can have non-static block as shown below in a example.

```
class NonStaticBlock1
{
	NonStaticBlock1(int i)
	{
		System.out.println("NonStaticBlock1("+i+")");
	}
	void fn(int a)
	{
		System.out.println("fn("+a+")");
	}
}
class NonStaticBlock2
{
	NonStaticBlock1 st1;
	NonStaticBlock1 st2;
	{
		st1 = new NonStaticBlock1(1);
		st2 = new NonStaticBlock1(2);
	}
	NonStaticBlock2()
```

```
	{
		System.out.println("NonStaticBlock2");
	}
}
public class NonStaticBlock {
	public static void main(String[] args)
	{
		NonStaticBlock2 nsb2 = new NonStaticBlock2();
	}
}
Output:
NonStaticBlock1(1)
NonStaticBlock1(2)
NonStaticBlock2
```

Above we can see that a non-static block starts with '{' and only difference is its not attached with a static keyword. Also we can see the non-static block will be executed before constructors.

➔ If we replace above programme as shown below then output will be different.

```
package ch5InitializationNCleanUp;

class StaticBlock1
{
	StaticBlock1(int i)
	{
		System.out.println("StaticBlock1("+i+")");
	}
	void fn(int a)
	{
		System.out.println("fn("+a+")");
	}
}

class StaticBlock2
{
	static StaticBlock1 st1;
	static StaticBlock1 st2;
	static StaticBlock1 st3;
	static StaticBlock1 st4;

	{
		st1 = new StaticBlock1(1);
		st2 = new StaticBlock1(2);
	}


	static
		{
			st3 = new StaticBlock1(3);
			st4 = new StaticBlock1(4);
		}

	StaticBlock2()
	{
		System.out.println("StaticBlock2");
	}
```

```
}

public class StaticBlock {
	public static void main(String[] args)
	{
		System.out.println("main method");
		StaticBlock2 st2 = new StaticBlock2();
		StaticBlock2.st3.fn(1);//an static member can be accessed through class
					name.
	}

}
Output:
main method
StaticBlock1(3)
StaticBlock1(4)
StaticBlock1(1)
StaticBlock1(2)
StaticBlock2
fn(1)
```

Above we could see that initialization of static block takes first before initialization of individual static object references.

Initialization takes place in a following way

➔ Static variables(both object reference and primitive variables, it is first initialized because it is connected to class itself.

➔ Non-static variables(both primitive variables and object references).

➔ Finally constructors will be called from base to derived classes.

➢ **Array Initialization** :

➔ One can take array of type **Object** as a argument list which is also called in C as 'variable argument list' also called as varargs. Varargs include unknown quantities of arguments and as well as unknown types. Example is shown below.

```
class VarArgsClass1{}

public class VarArgsClass
	{
	static void printArray(Object [] args)
	{
		for(Object obj : args)
		System.out.println(obj);
		System.out.println();
	}
	public static void main(String [] args)
	{
	printArray(new Object[]{new Integer(1), new Integer(2), new Integer(3)});
	printArray((Object[])new Integer[]{1,2,3,4});//this works too
	printArray(new Object[]{"one", "two", "three"});
```

```
      printArray(new Object[]{new VarArgsClass1(),new VarArgsClass(),
                        new VarArgsClass1()});
      //printArray();//This cant be used here.
      }
}
Output:
1
2
3


1
2
3
4

one
two
three

ch5InitializationNCleanUp.VarArgsClass1@e5b723
ch5InitializationNCleanUp.VarArgsClass@15a8767
ch5InitializationNCleanUp.VarArgsClass1@6f7ce9
```

➔ With Java SE5 now we can use ellipses to define a variable argument list or varargs. Example for the same is shown below.

```
class VarArgsClassEllipse1{}

public class VarArgsClassEllipse {
      static void printArray1(Object...args)
      {
            for(Object obj : args)
            System.out.println(obj);
            System.out.println();
      }
      public static void main(String[] args) {
            printArray1(new Integer(1), new Integer(2), new Integer(3));
            printArray1("one", "two", "three");
            printArray1(1.1, 2.2, 3.3);
            printArray1(new VarArgsClassEllipse1(), new VarArgsClassEllipse1()
            ,new VarArgsClassEllipse1());
            printArray1((Object[])new Integer[]{1,2,3,4});
            printArray1();
      }
}
Output:
1
2
3

one
two
three

1.1
2.2
```

```
3.3

ch5InitializationNCleanUp.VarArgsClassEllipse1@82ba41
ch5InitializationNCleanUp.VarArgsClassEllipse1@923e30
ch5InitializationNCleanUp.VarArgsClassEllipse1@130c19b

1
2
3
4
```

From above we can see that how using ellipses we can pass the arguments. Also in the program last but one printArray() method is using argument as autoboxing therefore it needed to be cast with Object so that compiler won't give error.

➔ Also in the previous example we can see that the last printArray() method didn't sent any argument at all but it is still working with Ellipses i.e. we can pass zero argument to the Ellipses. This will be useful when one will have optional trailing argument. Example for it is shown below.

```
public class OptionalTrailingArguments {
static void f(int required, String... trailing) {
System.out.print("required: " + required + " ");
for(String s : trailing)
System.out.print(s + " ");
System.out.println();
}
public static void main(String[] args) {
f(1, "one");
f(2, "two", "three");
f(0);
}
} /* Output:
required: 1 one
required: 2 two three
required: 0
```

➔ I above example one cannot use f(String...training, int required) then compiler will give error that " The variable argument type String of the method 'f' should be the last parameter. I.e. when there are two parameters ellipse or varargs should come after primitive type.

➔ Also there cannot be two varargs together like f(char...arg1, string...arg2)

Chapter 6. Overloaded and Overridden Methods

1) Overridden Methods :

The rules for overriding a method are as follows:

- The *argument list must exactly match* that of the overridden method.
- The *return type must exactly match* that of the overridden method.
- The *access level must not be more restrictive* than that of the overridden method.
- The *access level can be less restrictive* than that of the overridden method.
- The overriding method *must not throw new or broader checked exceptions* 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.
- *If a method can't be inherited, you cannot override it*.

2) Example for overriding methods :

Let's take a look at overriding the eat() method of Animal:

```
public class Animal {
public void eat() { }
}
```

Below table shows the illegal overrides of the method eat() in a subclass.

Illegal Override Code	Problem with the Code
`private void eat() { }`	Access modifier is more restrictive
`public void eat() throws IOException { }`	Declares a checked exception not declared by superclass version
`public void eat(String food) { }`	A legal overload, not an override, because the argument list changed
`public String eat() { }`	Not an override because of the return type, but not an overload either because there's no change in the argument list

3) Overloaded Methods :

Rules for overloading a methods are as below

- 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.

4) Example for Overloading methods

Example 1:

```
public class Animal {
public void eat() {
System.out.println("Generic Animal Eating Generically");
}
}
public class Horse extends Animal {
public void eat() {
System.out.println("Horse eating hay ");
}
public void eat(String s) {
System.out.println("Horse eating " + s);
}
}
```

Notice that the Horse class has both overloaded *and* overridden the eat() method. Table below shows which version of the three eat() methods will run depending on how they are invoked.

Example 2:

```
package OverloadingnArrays;

public class OverloadedCls {

	void meth(int i)
	{
		System.out.println("Integer value is "+i);
	}
	char [] meth(String str)
	{
		return str.toCharArray();
	}
```

```
	private void meth()
	{
		System.out.println("private meth");
	}
	void meth(int i,int j)
	{
		System.out.println("two args "+i+" & "+j);
	}

	public static void main(String[] args)
	{
		OverloadedCls oc = new OverloadedCls();
		oc.meth(10);
		System.out.println(oc.meth("java"));
		oc.meth(5, 15);
		oc.meth();
	}
}
Output:
Integer value is 10
java
two args 5 & 15
private meth
```

In the above example we can see how method arguments, return type, access control changes works with overloaded methods.

Method Invocation Code	Result
`Animal a = new Animal(); a.eat();`	`Generic Animal Eating Generically`
`Horse h = new Horse(); h.eat();`	`Horse eating hay`
`Animal ah = new Horse(); ah.eat();`	`Horse eating hay` Polymorphism works—the actual object type (Horse), not the reference type (Animal), is used to determine which `eat()` is called.
`Horse he = new Horse(); he.eat("Apples");`	`Horse eating Apples` The overloaded `eat(String s)` method is invoked.
`Animal a2 = new Animal(); a2.eat("treats");`	Compiler error! Compiler sees that Animal class doesn't have an `eat()` method that takes a String.
`Animal ah2 = new Horse(); ah2.eat("Carrots");`	Compiler error! Compiler *still* looks only at the reference type, and sees that Animal doesn't have an `eat()` method that takes a string. Compiler doesn't care that the actual object might be a Horse at runtime.

Example 2

```
class BaseClass
{
	int a=5,b=5;
	int sumInt()
	{
		int sum = 0;
		sum = a+b;
		return sum;
	}
```

```
	void printSum(BaseClass obj)
	{
		System.out.println("this is base class method "+obj.sumInt());
	}
}

class DerivedClass extends BaseClass
{
	int i=10,j=10;
	int sumInt()
	{
		int sum = 0;
		sum = i+j;
		return sum;
	}
}

public class PolymrphClass
{
	public static void main(String[] args)
	{
		// TODO Auto-generated method stub
		BaseClass bc = new BaseClass();
		DerivedClass dc = new DerivedClass();
		System.out.println(bc.sumInt());
		System.out.println(dc.sumInt());

		bc.printSum(dc);

		bc = dc;
		System.out.println(bc.sumInt());
	}
}
Output:
10
20
this is base class method 20
20
```

5) Difference between overloaded and overridden methods are as follows

	Overloaded Method	Overridden Method
argument list	Must change	Must not change
return type	Can change	Must not change
exceptions	Can change	Can reduce or eliminate. Must not throw new or broader checked exceptions
access	Can change	Must not make more restrictive (can be less restrictive)
invocation	*Reference* type determines which overloaded version (based on declared argument types) is selected. Happens at *compile* time. The actual *method* that's invoked is still a virtual method invocation that happens at runtime, but the compiler will already know the *signature* of the method to be invoked. So at runtime, the argument match will already have been nailed down, just not the actual *class* in which the method lives.	*Object* type (in other words, *the type of the actual instance on the heap*) determines which method is selected. Happens at *runtime*.

Access Modifiers :

1) There are three access modifiers *public*, *private* and *protected* but there are four access control *default* being the fourth.

2) A class could have only 2 out of 4 access control. They are *public* and *default.*

Default access : Its a package level access and any class A in package x cannot be a super class of a class B in package Y. i.e. default modifier makes class invisible outside the package. When no modifier is used for a class then it considered as *default* modifier.

Ex: `class Beverage {}` has default access.

Public access : When a class is marked with *public* modifier then it is available to all the classes virtually in the java universe.

Ex: `public class Beverage {}`

3) Member Access :

- All the four access controls will be used by class members.
- It is always better to check class modifier before member modifiers for faster understanding.
- public : When a member variable or method is declared public then it means all other classes regardless of package have access to it(assuming the class is *visible* outside package).

Ex :

```
package book;
import cert.*; // Import all classes in the cert package
class Goo {
public static void main(String [] args) {
Sludge o = new Sludge();
o.testIt();
}
}
```

Now look at the second file

```
package cert;
public class Sludge {
public void testIt() {
System.out.println("sludge");
}
}
```

Also when a member of a super class is declared *public* then subclass will inherits that member regardless of both classes are in the same package. Hence (.) operator i.e. reference is not needed to use those members. In above example testIt() method could be directly called.

Example

```
class InheritorMethCheck1
{
      void meth1()
      {
            System.out.println("Inside meth1");
      }
}

class InheritorMethCheck2 extends InheritorMethCheck1
{
      void meth2()
      {
            meth1();
            System.out.println("Inside meth2");
      }
}

public class InheritorMethCheck {

      public static void main(String[] args)
      {
            InheritorMethCheck2 imc = new InheritorMethCheck2();
            imc.meth2();
      }
}
Output:
Inside meth1
Inside meth2
```

- Private :

Members marked *private* could be used by only the class in which they are defined, for other classes they are invisible.

- Ex :

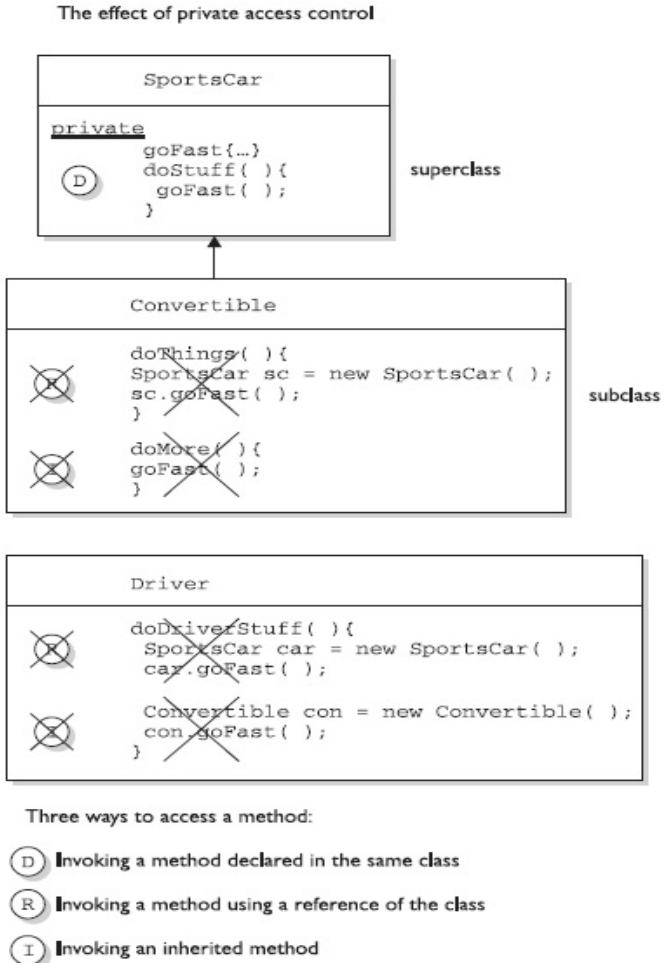
```
package chapter2;

class PrivateMemberClass1
{
  private int a=10, b=10;
  int c = 20;
  private int addint()
  {
        int sum = 0;
        sum = a+b;
        return sum;
  }
  int mulint()
  {
        int mul;
        mul=a*b;
        return mul;
  }
}
public class PrivateMemberClass {
```

```
    /*Private members cannot be used outside the same class */
    public static void main(String[] args) {
          PrivateMemberClass1 prcls = new PrivateMemberClass1();
          //System.out.println(prcls.addint());//dont compile
          //System.out.println(prcls.a); //dont compile
          System.out.println(prcls.c);
          System.out.println(prcls.mulint());
    }
}
```

If we try to remove comment line for method addint() then compiler will give following error.

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
    The method addint() from the type PrivateMemberClass1 is not visible
    at chapter2.PrivateMemberClass.main(PrivateMemberClass.java:24)
```

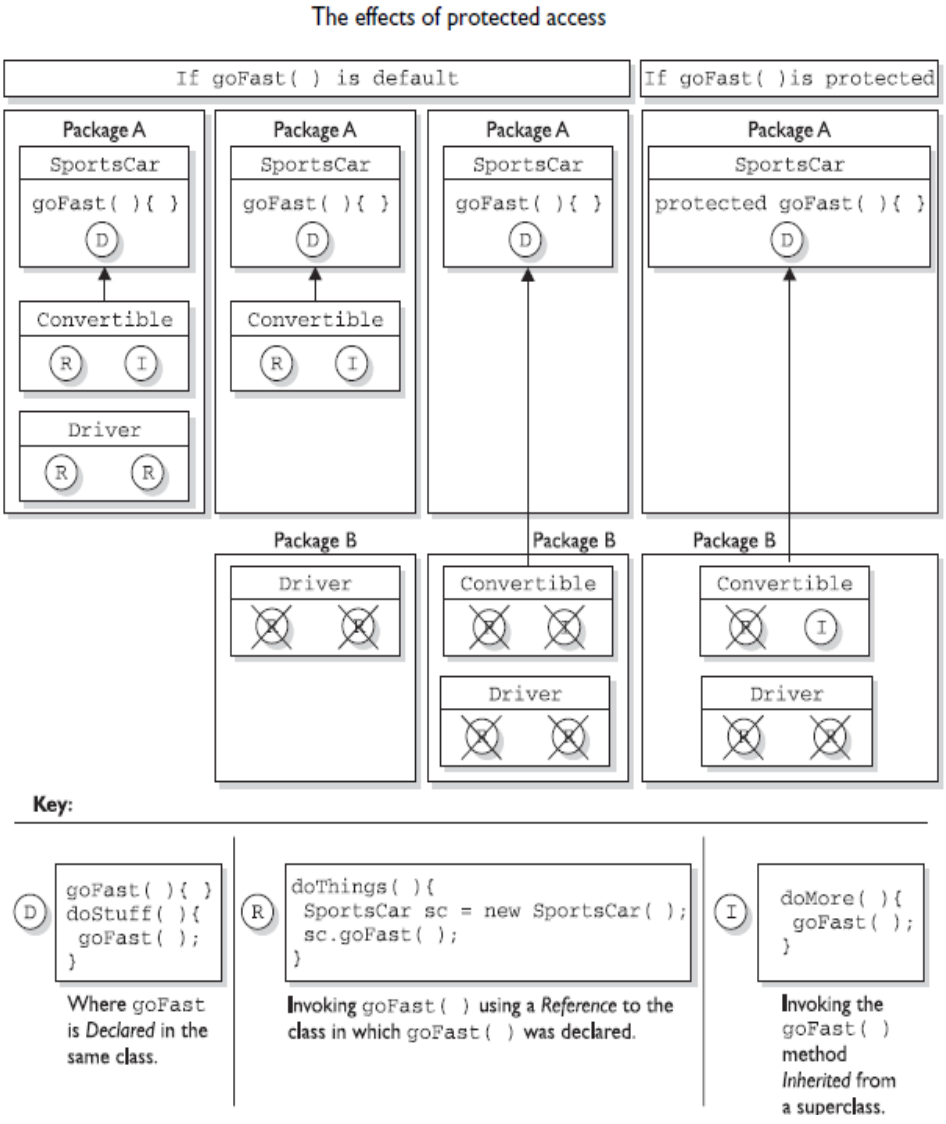
- Even for a subclass, superclass private members are invisible. When subclass declares a method of the same name following all the rules of overriding even then the method wont considered overridden as superclass method is invisible.
- It is always better to mark member variables as private and give access to them through public methods. For example if cat class have variable weight then it could take a negative value hence giving access through methods is a better solution where one can make a check for valid values.
- Effect of public and private modifiers on classes from same or diff packages is shown below.



- Protected and Default :

  protected and *default* modifiers are same with only one difference. *default* modifier is only for package level and no class outside package access those members by reference or inheritance. But with *protected* modifier a class in one package can inherit(that means only through subclass and not be reference i.e. (.) operator) protected members of another package. Hence one can say default is only for package level but protected is for package+kids.

- If Class A of package 1 has protected members and if it has been extended by Class B of package 2 then Class B will have access to protected members of Class A. But what if Class C in package 2 extends Class B of same package. Does Class C will also have access to protected members of Class A. NO. Because protected members in subclass becomes private members of it and wont be visible to other subclasses.
- The effect of default and protected access is shown below



- Summary :

Determining Access to Class Members

Visibility	Public	Protected	*Default*	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From any non-subclass class outside the package	Yes	No	No	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes	No	No

Chapter 7: Reusing Classes

➔ In case of inheritance we call base class methods from derived class using keyword 'super'. Ex : super.fn().

➔ Compiler put default constructor for each class but when one want to have two constructor with parameter and without parameter, Also when he want to call both of them then he have to explicitly create a default(non-para) constructor.

➔ We already know that when there are more than one constructor present then we can use 'this' keyword to call one constructor from another. In case of inheritance we know that when we create an object of a derived class then super class constructor(default or implicit) will be called automatically. But if user has declared a parameterized constructor and have not declared default constructor separately then there will be error. This error can be avoided by calling parameterized super class constructor using keyword **super** in derived class constructor. Example of these is show below.

```
package ch7ReusingClasses;

class SuperConstructor1
{
	SuperConstructor1(int i)
	{
		System.out.println("inside supcon1");
	}
}

class SuperConstructor2 extends SuperConstructor1
{
	SuperConstructor2(int j)
	{
		super(j);
		System.out.println("inside supcon2");
	}
}

public class SuperConstructor extends SuperConstructor2 {
	/**How super keyword is used inside constructors */
	SuperConstructor(int k)
	{
		super(11);
		System.out.println("inside supcon");
	}
	public static void main(String[] args) {
		SuperConstructor sp = new SuperConstructor(11);
	}
}
Output:
inside supcon1
inside supcon2
inside supcon
```

Note: If we remove super(j) or super(11) in derived class constructor then compiler give error "implicit super constructor SuperConstructor1()(or SuperConstructor2()) undefined. Must explicitly invoke another constructor".

➔ Also one can avoid using ‘super’ if default constructors are defined by programmer.

➔ We k now that inside a same class with many constructors we use ‘this’ to reuse the code and same can be done in inheritance through ‘super’.

➔ (KnS) When a overridden subclass method wants to use functionality of superclass method and wants to add some extra functionality then this could be done using keyword ‘super’ as shown below.

```
package ch7ReusingClasses;

class Animal
{
	void animalWalk()
	{
		System.out.println("All animal walks");
	}
}

class Wolf extends Animal
{
	void animalWalk()
	{
		super.animalWalk();
		System.out.println("Wolf crowls too");
	}
}

public class BaseMethodUsingSuper
{
	public static void main(String[] args)
	{
		Wolf w = new Wolf();
		w.animalWalk();
	}
}
Output:
All animal walks
Wolf crowls too
```

➔ When a base class method is overloaded several times in derived class then redefining that method in derive class will not hide any base class versions. Example

```
package ch7ReusingClasses;

class NameHidMethodCls1
{
	char doh(char c)
	{
		System.out.println("char(doh)");
		return 'a';
	}
	float doh(float i)
	{
		System.out.println("float(doh)");
		return 1.0f;
```

```
	}
}

class NameHidMethodCls2 {}

class NameHidMethodCls3 extends NameHidMethodCls1
{
	void doh(NameHidMethodCls2 nms2)
	{
		System.out.println("nms2(doh)");
	}
}

public class NameHidMethodCls {
	/** How derive class have access to all overloaded
	 * version of methods */
	public static void main(String[] args) {
		NameHidMethodCls3 nms3 = new NameHidMethodCls3();
		nms3.doh(1);
		nms3.doh('a');
		nms3.doh(1.0f);
		nms3.doh(new NameHidMethodCls2());
	}
}
```

Output:

```
float(doh)
char(doh)
float(doh)
nms2(doh)
```

➔ Also one will see that it's far more common to **override** methods of the same name, using exactly the same signature and return type as in the base class. Java SE5 added a new annotation @Override which prevents methods to get **overloaded.** Example

```
class Lisa extends Homer
 {
    @Override void doh(Milhouse m)
    {
       System.out.println("doh(Milhouse)");
    }
}
```

➔ When one has to chose between composition or inheritance it depends on many criteria. But if 'upcasting' is involved then one should use inheritance else one can decide on what to do.

➢ **<u>Order of initialization with inheritance</u> :**

```
package ch7ReusingClasses;

class BaseOrderOfInitilization1
{
	private int i = 9;
	protected int j;
	BaseOrderOfInitilization1()
```

```
	{
		System.out.println("i="+i+" "+"j="+j);
		j=39;
	}
	private static int x1 = printInit("static BaseOrdInit1.x1 initialized");
	static int printInit(String s)
	{
		System.out.println(s);
		return 47;
	}
}

class OrderOfInitialization extends BaseOrderOfInitilization1
{
	private int k = printInit("OrdInt is initialized");
	OrderOfInitialization()
	{
		System.out.println("k="+k);
		System.out.println("j="+j);
	}
	private static int x2 = printInit("static Ordint.x2 initialized");

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		OrderOfInitialization oi = new OrderOfInitialization();
	}
}
Output:
static BaseOrdInit1.x1 initialized
static Ordint.x2 initialized
i=9 j=0
OrdInt is initialized
k=47
j=39
```

➔ (*)From above we could see that the order of initialization takes place as shown below

1. Static reference variables
2. Static primitive variables
3. Non-static reference variables
4. Non-static primitive variables
5. Constructors

Now above execution takes place separately in two different scenerio

A. With separate class : Execution take place from 1 to 5 class wise.
B. With Inheritance : With Inheritance execution from 1 to 2 take place from Base to derive and then 3 to 5 take place class wise.

➔ Above can also be explained with the example given below

```
package ch8Polymorphism;

class OrderOfInitForVarNObjVar1

{
	OrderOfInitForVarNObjVar1()
	{
```

```
	}
	OrderOfInitForVarNObjVar1(int a)
	{
		System.out.println("OrderOfInitForVarNObjVar1 "+a);
	}
}

class OrderOfInitForVarNObjVar2 extends OrderOfInitForVarNObjVar1
{
	static OrderOfInitForVarNObjVar1 ov1 = new OrderOfInitForVarNObjVar1(1);

	OrderOfInitForVarNObjVar1 ov2 = new OrderOfInitForVarNObjVar1(2);
}

class OrderOfInitForVarNObjVar3 extends OrderOfInitForVarNObjVar2
{
	static OrderOfInitForVarNObjVar1 ov3 = new OrderOfInitForVarNObjVar1(3);

	OrderOfInitForVarNObjVar1 ov4 = new OrderOfInitForVarNObjVar1(4);
}

public class OrderOfInitForVarNObjVar {
	public static void main(String[] args)
	{
		OrderOfInitForVarNObjVar3 ovn1 = new OrderOfInitForVarNObjVar3();

	}
}
Output:
OrderOfInitForVarNObjVar1 1
OrderOfInitForVarNObjVar1 3
OrderOfInitForVarNObjVar1 2
OrderOfInitForVarNObjVar1 4
```

Note: In above example default constructor will be executed for `OrderOfInitForVarNObjVar1` class.

Chapter 8 : Polymorphism

➔ We have seen in last chapter that in inheritance allows many types(inherited from the same base type) to be treated as if they were one type. The polymorphism allows one type to express its distinction from another, similar type, as long as they derived from the same base class. This distinction is expressed through differences in behaviour of the methods that you can call through the base class

➔ We know that in cases where upcasting has been used, java works fine because of late binding. That is compiler doesn't know which piece of code will be first executed during compile time it is only got decided during run time through late binding. This late binding is also called as dynamic binding. Polymorphism is possible because of this.

➔ Following is an example of polymorphism and also shows how late binding works.

```
package ch8Polymorphism;

class Instrument
{
	void play(){System.out.println("instrument play()");}
	String what(){return "instrument" ;}
}
class Wind extends Instrument
{
	void play(){System.out.println("wind play()");}
	String what(){return "wind" ;}
}
class Percussion extends Instrument
{
	void play(){System.out.println("percussion play()");}
	String what(){return "percussion" ;}
}
class Woodwind extends Wind
{
	void play(){System.out.println("woodwind play()");}
	String what(){return "woodwind" ;}
}
class Brass extends Wind
{
	void play(){System.out.println("brass play()");}
	String what(){return "brass" ;}
}

public class MusicPolymorphism
{
	public static void tune(Instrument i)
	{
		i.play();
	}
public static void tuneAll(Instrument[] I)
{
	for(Instrument i : I)
		tune(i);
}
public static void main(String [] args)
```

```
{
	Instrument [] inst = {
			new Wind(),
			new Percussion(),
			new Woodwind(),
			new Brass(),
	};
	tuneAll(inst);
}
}
Output :
wind play()
percussion play()
woodwind play()
brass play()
```

➔ **Pitfall : Overriding private method:**

See the programme given below

```
package ch8Polymorphism;

public class PrivateOverride
{
	private void f(){ System.out.println("private f()"); }

	public static void main(String[] args)
	{
		PrivateOverride pv = new Derived();
		pv.f();
	}
}

class Derived extends PrivateOverride
{
	public void f(){System.out.println("public f()");}
}
Output:
private f()
```

In the above example one may expect output as “public f()” . But a private method is automatically final and hidden from derived class. In the above case public method f() is a brand new method, its not even overloaded since base class version of f() is not visible in derived class. **The point is where there is no overriding polymorphism doesn’t work there**. i.e. Virtual Method Invocation(VMI) only works when there is overridding. Example which shows that polymorphism doesnot work with overloaded methods.

```
package ch8Polymorphism;

class NNoOverrideNoPoly1
{
	void meth1(String str)
	{
		System.out.println(str);
	}
}
```

```
class NNoOverrideNoPoly2 extends NNoOverrideNoPoly1
{
	void meth1(String str1, String str2)
	{
		System.out.println(str1);
		System.out.println(str2);
	}
}

class NNoOverrideNoPoly3 extends NNoOverrideNoPoly1
{
	void meth1(String str1, String str2, String str3)
	{
		System.out.println(str1);
		System.out.println(str2);
		System.out.println(str3);
	}
}

public class NoOverrideNoPoly
{
	public static void main(String[] args)
	{
		NNoOverrideNoPoly1 nonp1 = new NNoOverrideNoPoly1();
		NNoOverrideNoPoly2 nonp2 = new NNoOverrideNoPoly2();
		NNoOverrideNoPoly3 nonp3 = new NNoOverrideNoPoly3();

		nonp1 = nonp2;
		nonp1.meth1("one and two");
		//nonp1.meth1("one","two");
		//nonp2 meth1() method is not same as nonp1 meth1() method hence
hidden.

		nonp1 = nonp3;
		nonp1.meth1("one and three");
		//nonp1.meth1("one","two","three");
		//nonp3 meth1() method is not same as nonp1 meth1() method hence
hidden.
	}
}
Output:
one and two
one and three
```

➔ **Pitfall : Fields and static methods** :

Also polymorphism is applied for only methods and not for fields and static methods. As shown in following example.

```
package ch8Polymorphism;

class FieldBase
{
	public int field = 0;
	public int getField()
	{
		return field;
```

```
	}
}

class FieldDerive extends FieldBase
{
	public int field = 1;
	public int getField()
	{
		return field;
	}
	public int getSuperField()
	{
		return super.field;
	}
	}

public class FieldAccessPoly {
	public static void main(String[] args)
	{
		FieldBase fb = new FieldDerive();
		System.out.println("sub.field = "+fb.field+" sub.getfield() = "+
				fb.getField());
		FieldDerive fd = new FieldDerive();
		System.out.println("sub.field = "+fd.field+" sub.getfield() = "+
				fd.getField()+" super.field = "+fd.getSuperField());
	}

}
Output:
sub.field = 0 sub.getfield() = 1
sub.field = 1 sub.getfield() = 1 super.field = 0
```

When a **Sub** object is upcast to a **Super** reference, any field accesses are resolved by the compiler, and are thus not polymorphic. I.e. **things which can be resolved during compile time polymorphism doesn't work there**.

```
package ch8Polymorphism;

class StaticBase
{
	public static String staticGet()
	{
		return "Base staticGet()";
	}
	public String dynamicGet()
	{
		return "Base dynmaicGet()";
	}
}

class StaticDerive extends StaticBase
{
	public static String staticGet()
	{
		return "Derive staticGet()";
	}
```

```
	public String dynamicGet()
	{
		return "Derive dynamicGet()";
	}
}

public class StaticMethodPolymorphism
{
	public static void main(String[] args)
	{
		StaticBase sb = new StaticDerive();
		System.out.println(sb.staticGet());
		System.out.println(sb.dynamicGet());
	}
}
Output:
Base staticGet()
Derive dynamicGet()
```

Static methods are associated with the class and not with the individual objects.

➔ **From above we can note that Polymorphism doesn't work with Fields, private methods and static methods**.

➔ How construction takes place with polymorphism. Example given below.

```
package ch8Polymorphism;

class Meal
{
	Meal() { System.out.println("Meal()"); }
}
class Bread
{
	Bread() { System.out.println("Bread()"); }
}
class Cheese
{
	Cheese() { System.out.println("Cheese()"); }
}
class Lettuce
{
	Lettuce() { System.out.println("Lettuce()"); }
}
class Lunch extends Meal
{
	Lunch() { System.out.println("Lunch()"); }
}
class PortableLunch extends Lunch
{
	PortableLunch() { System.out.println("PortableLunch()");}
}

public class ConstructorPolymorph extends PortableLunch
{
	private Bread b = new Bread();
	private Cheese c = new Cheese();
	private Lettuce l = new Lettuce();
```

```
	private Meal ml = new Lunch();
	public ConstructorPolymorph()
	{
		System.out.println("ConstructorPolymorph()");
	}
	public static void main(String[] args)
	{
	new ConstructorPolymorph();
	}
}
Output:
Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Meal()
Lunch()
ConstructorPolymorph()
```

➔ **Inheritance and cleanup** :

When one have clean up issue that is if one want to handle clean up thing apart from Garbage collector then one must create a method which will dispose things. Lets say the method name is dispose() and following is a example of how it should work.

```
package ch8Polymorphism;

class Characteristic {
private String s;
Characteristic(String s) {
this.s = s;
System.out.println("Creating Characteristic " + s);
}
protected void dispose() {
System.out.println("disposing Characteristic " + s);
}
}
class Description {
private String s;
Description(String s) {
this.s = s;
System.out.println("Creating Description " + s);
}
protected void dispose() {
System.out.println("disposing Description " + s);
}
}
class LivingCreature {
private Characteristic p =
new Characteristic("is alive");
private Description t =
new Description("Basic Living Creature");
LivingCreature() {
System.out.println("LivingCreature()");
}
protected void dispose() {
```

```
System.out.println("LivingCreature dispose");
t.dispose();
p.dispose();
}
}
class Animal extends LivingCreature {
private Characteristic p =
new Characteristic("has heart");
private Description t =
new Description("Animal not Vegetable");
Animal() { System.out.println("Animal()"); }
protected void dispose() {
System.out.println("Animal dispose");
t.dispose();
p.dispose();
super.dispose();
}
}
class Amphibian extends Animal {
private Characteristic p =
new Characteristic("can live in water");
private Description t =
new Description("Both water and land");
Amphibian() {
System.out.println("Amphibian()");
}
protected void dispose() {
System.out.println("Amphibian dispose");
t.dispose();
p.dispose();
super.dispose();
}
}
public class FrogDispose extends Amphibian {
private Characteristic p = new Characteristic("Croaks");
private Description t = new Description("Eats Bugs");
public FrogDispose() { System.out.println("Frog()"); }
protected void dispose() {
System.out.println("Frog dispose");
t.dispose();
p.dispose();
super.dispose();
}
public static void main(String[] args) {
FrogDispose frog = new FrogDispose();
System.out.println("Bye!");
frog.dispose();
}
}
Output:
Creating Characteristic is alive
Creating Description Basic Living Creature
LivingCreature()
Creating Characteristic has heart
Creating Description Animal not Vegetable
Animal()
Creating Characteristic can live in water
Creating Description Both water and land
Amphibian()
```

```
Creating Characteristic Croaks
Creating Description Eats Bugs
Frog()
Bye!
Frog dispose
disposing Description Eats Bugs
disposing Characteristic Croaks
Amphibian dispose
disposing Description Both water and land
disposing Characteristic can live in water
Animal dispose
disposing Description Animal not Vegetable
disposing Characteristic has heart
LivingCreature dispose
disposing Description Basic Living Creature
disposing Characteristic is alive
```

In the above example we can see that in each class in the hierarchy contains a member object of type Characteristics and Description. The order of disposal should be reverse of the order of initialization. For base classes one should perform the derived class clean up first, then base class clean up. From output one can see that all parts of the Frog object are disposed in reverse order of creation.

➔ In some of the cases we need to call a method from a constructor. In case of overridden methods the output may be completely different as shown in the example below.

```
package ch8Polymorphism;

class ConstructorMethod1
{
	void draw()
	{
		System.out.println("ConstructorMethod1.draw()");
	}
	ConstructorMethod1()
	{
		System.out.println("ConstructorMethod1 const starts");
		draw();
		System.out.println("ConstructorMethod1 const ends");
	}
}

class ConstructorMethod2 extends ConstructorMethod1
{
	int radius = 1;
	ConstructorMethod2(int r)
	{
		radius = r;
		System.out.println("ConstructorMethod2 radius "+radius);
	}
	void draw()
	{
		System.out.println("ConstructorMethod2.draw.radius "+radius);
	}
}
```

```
public class ConstructorMethod
{
	public static void main(String[] args)
	{
		ConstructorMethod2 cs2 = new ConstructorMethod2(5);
	}
}
Output: ConstructorMethod1 const starts
ConstructorMethod2.draw.radius 0
ConstructorMethod1 const ends
ConstructorMethod2 radius 5
```

In the above example we can see that draw() is overridden in the derived class. But when draw() is called in ConstructorMethod1s constructor the derived class draw() method has been called. This is the bug very tough to find. Also We can see the above output because of following order of initialization.

a. The storage allocated for the object is initialized to binary zero before anything else happens.

b. The base-class constructors are called as described previously. At this point, the overridden **draw( )** method is called (yes, *before* the **ConstructorMethod2** constructor is called), which discovers a **radius** value of zero, due to Step 1.

c. Member initializers are called in the order of declaration

d. The body of the derived-class constructor is called.

Because of this its always safe to call only final and private(which are implicitly final) methods of base class in constructors because they cant be overridden.

➔ **Covariant return types :**

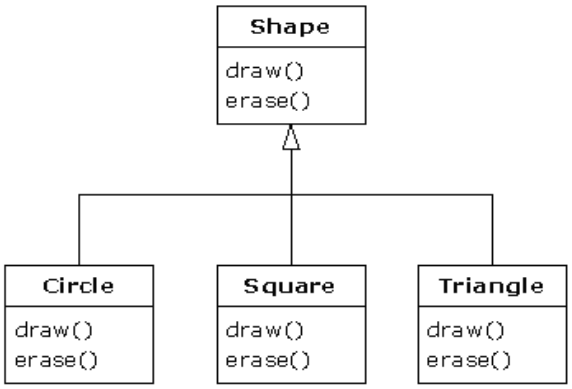
Java SE5 adds the covariant return types, which means that an overridden method in a derived class can return a type which is derived from the type returned by the base class. Example is given below.

```
class Grain {
public String toString() { return "Grain"; }
}
class Wheat extends Grain {
public String toString() { return "Wheat"; }
}
class Mill {
Grain process() { return new Grain(); }
}
class WheatMill extends Mill {
Wheat process() { return new Wheat(); }
}
public class CovariantReturn {
public static void main(String[] args) {
Mill m = new Mill();
Grain g = m.process();
System.out.println(g);
```

```
m = new WheatMill();
g = m.process();
System.out.println(g);
}
} /* Output:
Grain
Wheat
*///:~
```

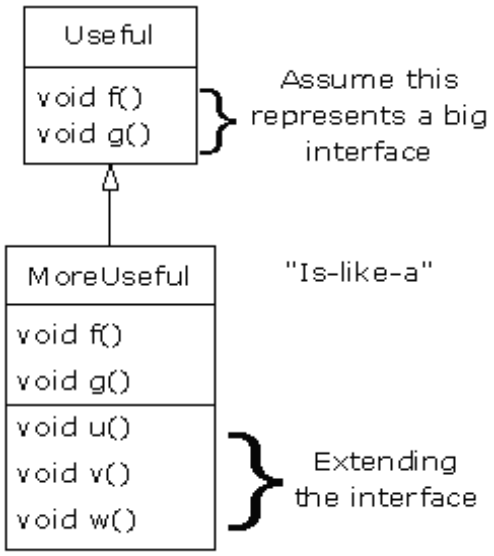
Earlier java versions would have forced overridden method process() to return 'Grain' and not wheat.

➔ When inheritance is used and if its pure inheritance(is-a relationship) that is both Base and Derived class have same interfaces then its good. Pure inheritance diagram is shown below.



With pure inheritance base class can receive any message that one can send to derive class(through upcasting, polymorphism) because both have the same interfaces.

But when tries to add more interfaces to the derived class as show below.



Then the extended part of the interface in derived class is not available from the base class, so once you upcast you cant call the add interfaces. A program is shown below.

```
package ch8Polymorphism;

class ExtendedMethods1
{
	int a;
	void method1()
	{
		System.out.println("Base method1");
	}
	void method2()
	{
		System.out.println("Base method2");
	}
}

class ExtendedMethods2 extends ExtendedMethods1
{
	void method1()
	{
		System.out.println("Derived method1");
	}
	void method2()
	{
		System.out.println("Derived method2");
	}
	void method3()
	{
		System.out.println("Derived method3");
	}
}

public class ExtendedMethods {

	public static void main(String[] args)
	{
		ExtendedMethods1 em1 = new ExtendedMethods1();
		ExtendedMethods2 em2 = new ExtendedMethods2();
		em1.method1();
		em1.method2();

		em1=em2;

		em1.method1();
		em1.method2();
		//em1.method3();//cant call method3 cos its undefined in base class
	}
}
Output:
Base method1
Base method2
Derived method1
Derived method2
```

Above we can see em1.method3() has been commented out because method3() is not present in the base class hence compiler throws error.

➔ To solve the above problem we have a option of 'Downcasting'. As we know upcasting is always safe as we are sure that there cant be bigger interfaces compare to base class but downcasting is not that safe as you don't know that a shape (for example) is actually a circle. It can be triangle or square or some other type.

To be ensure that downcast is safe java performs checks on every cast. During runtime downcast will be checked and if there is a type mismatch then java throws an **ClassCastException.** This act of checking type at run type is called 'runtime type identification' (**RTTI**). Follwing program demonstrate when java does that.

➔ Example : 1

```
package ch8Polymorphism;

class ExtndMethDowncast1
{
	void emdMethod1()
	{
		System.out.println("base method 1");
	}
	void emdMethod2()
	{
		System.out.println("base method 2");
	}
}

class ExtndMethDowncast2 extends ExtndMethDowncast1
{
	void emdMethod1()
	{
		System.out.println("derive method 1");
	}
	void emdMethod2()
	{
		System.out.println("derive method 2");
	}
	void emdMethod3()
	{
		System.out.println("derive method 3");
	}
}

public class ExtndMethDowncast {

	public static void main(String[] args)
	{
		// TODO Auto-generated method stub
		ExtndMethDowncast1 emd1 = new ExtndMethDowncast1();
		ExtndMethDowncast2 emd2 = new ExtndMethDowncast2();
		emd1 = emd2;
		emd1.emdMethod1();
		emd1.emdMethod2();
		((ExtndMethDowncast2)emd1).emdMethod3();
```

```
	}
}
Output :
derive method 1
derive method 2
derive method 3
```

➔ Example : 2

```
package ch8Polymorphism;

class ExtndMethodDowncasting1
{
	int a;
	void method1()
	{
		System.out.println("Base method1");
	}
	void method2()
	{
		System.out.println("Base method2");
	}
}

class ExtndMethodDowncasting2 extends ExtndMethodDowncasting1
{
	void method1()
	{
		System.out.println("Derived method1");
	}
	void method2()
	{
		System.out.println("Derived method2");
	}
	void method3()
	{
		System.out.println("Derived method3");
	}
}

public class ExtndMethodDowncasting {

	public static void main(String[] args)
	{
		ExtndMethodDowncasting1 [] em = { new ExtndMethodDowncasting1(), new
				ExtndMethodDowncasting2() };
		em[0].method1();
		em[1].method2();
		((ExtndMethodDowncasting2)em[1]).method3();//RTTI
		//((ExtndMethodDowncasting2)em[0]).method3();//Exception
		}
}
Output:
Base method1
Derived method2
Derived method3
```

Note that class cast has been used here i.e. ExtndMethodDowncasting2.

Chapter 9 : Interfaces

1. Abstract class :

➔ We know that sometime we do need a class which gives methods that can be expressed differently through its subtypes. Also its absurd to create objects of these classes so compiler gives an option so that objects of them can't be created. This can be done through 'Abstract' methods and a class with one or more abstract methods is qualified to be called as 'Abstract' class.

➔ (KnS)Instance of the abstract class cannot be created, it should be extended. Similarly abstract method is the one which should be overridden.

➔ (KnS)Abstract method is the one which are not been implemented. The abstract methods ends up with semicolon rather than braces.

➔ (KnS)Abstract class can have both abstract and non-abstract methods.

➔ (KnS)A class cannot be both abstract and final because they are exactly opposite of each other and code wont compile.

➔ If one inherit from an abstract class and if one want to create an object of the new derived class then first he has to implement all the abstract methods of abstract base class. If one don't then the new class will also be qualified as an Abstract class.

➔ When a concrete class implement abstract method then it should be marked public.

➔ (KnS) An abstract class can implement an interface. And an concrete class extending such abstract class should implement all the abstract methods.

Example : abstract class Ball implements Bounceable {}

```
package chapter2;//Kathy n Sierra

interface InterfaceClass1
{
      int a=10;
      void IntfMethod();
}

class InterfaceClass2 implements InterfaceClass1

{
      public void IntfMethod()
      {
            System.out.println("InterfaceClass2 abstract method");
      }
}

abstract class AbClass1 implements InterfaceClass1
{
      abstract void absMethod();
    //void printint();//Error
```

```
}
public class InterfaceClass extends InterfaceClass2 implements InterfaceClass1
{
	public static void main(String args[])
	{
		InterfaceClass2 ic = new InterfaceClass2();
		ic.IntfMethod();
		InterfaceClass ic1 = new InterfaceClass();
		ic1.absMethod();
		ic1.IntfMethod();
	}
	    public void absMethod()
	    {
			System.out.println("This is a abstract method");

	    }
	      public void IntfMethod()//overridden method
	      {
			System.out.println("this is interface method");
	      }
}
Output:
InterfaceClass2 abstract method
This is a abstract method
this is interface method
```

➔ One can have an Abstract class without any abstract methods if one want to have a class but don't want to create an object of that class. Example is shown below.

```
package ch8Polymorphism;

abstract class AbsWOAbsMethod1
{
	void method1()//methods is implementd hence not marked abstract
	{
		System.out.println("method1");
	}
	void method2()
	{
		System.out.println("method2");
	}
}

class AbsWOAbsMethod2 extends AbsWOAbsMethod1
{
	void methodAbs()
	{
		method1();
	}
}

public class AbsWOAbsMethod {

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

```
            //ClassAbs ab1 = new ClassAbs();//error:cant instanstiate
                                                        //the type ClassAbs
            AbsWOAbsMethod2 abs = new AbsWOAbsMethod2();
            abs.method1();
      }
}
Output:
Method1
```

Example : 2

```
package ch9Interfaces;

abstract class NoAbsMethClass1
{
      void method1()
      {
            System.out.println("method1");
      }

      void method2()
      {
            System.out.println("method2");
      }
}

abstract class NoAbsMethClass2 extends NoAbsMethClass1
{
      void method3()
      {
            System.out.println("method3");
      }
}

public class NoAbsMethClass extends NoAbsMethClass2
{
      public static void main(String[] args)
      {
            NoAbsMethClass na = new NoAbsMethClass();
            na.method1();
            na.method2();
            na.method3();
      }
}
Output:
method1
method2
method3
```

Note : Above we can see an abstract class can be extended by both an another abstract class and concrete class even if it don't have abstract method. Also it can be noted that a class with an abstract method has to be marked abstract but not vice versa i.e. an abstract class don't need to have an abstract method.

2. Interfaces :

➔ Interfaces are nothing but fully abstract classes as none of its methods have any bodies.

➔ (KnS) Interfaces have all the methods as abstract method i.e. if an abstract class have all the methods as abstract then it should be marked as interface rather.

➔ An Object for interface or abstract class cannot be created.

➔ Interface is created by using 'interface' keyword.

➔ Class needs to implement an interface and a class can implement any number of interfaces separated by commas. Example : class A implement interface1, interface2, interface3.

➔ (KnS) Interface has(implicitly)

- ➔ Interface: public and abstract
- ➔ Method : public and abstract
- ➔ Variable : public static final

➔ During implementation in a concrete class all methods of interface should be marked as 'public' else compiler will give error.

➔ For using an interface, a class should impelement it first.

Example : `Class A implement interface1`

- ➔ (KnS) An interface can extend one or more other interfaces.
- ➔ (KnS) Interface can't extend anything but other interfaces.
- ➔ (KnS) public abstract interface Rollable {}

  Public interface Rollable {}

Both are legal, but no need to mention 'abstract' in first statement as interfaces are implicitly abstract.

➔ (KnS) As variables in interface are public static final implicitly, so classes which implement them directly inherit these constant variable too.

➔ (KnS) An abstract class can implement an interface. And an concrete class extending such abstract class should implement all the abstract methods.

Ex. `abstract class Ball implements Bounceable {}`

➔ When a class(abstract or concrete) and an interface have the same method(concrete method inside class) name, return type and access specifier(public in this case) and if a subclass extends and implement them then its not mandatory to implement common method in subclass.

```
package ch9Interfaces;
```

```
interface InterfaceFight
{
	void fight();
}

interface InterfaceBright
{
	void bright();
}

interface InterfaceTight
{
	void tight();
}

/*abstract*/ class InterfaceClassShareMethod1
{
	public void fight()
	{
		System.out.println("InterfaceClassShareMethod1 class");

	}

	//abstract void abstmethod();
}

public class InterfaceClassShareMethod extends InterfaceClassShareMethod1
implements InterfaceFight, InterfaceBright, InterfaceTight
{
/**
	 * a class extends another class and an interface both
	 * of whom have the same method
	 */
	public void bright()
	{
		System.out.println("this is bright");
	}
	public void tight()
	{
		System.out.println("this is tight");
	}
	/*void abstmethod()
	{
		System.out.println("abstract method");
	}*/

	public static void main(String[] args) {
		InterfaceClassShareMethod icm=new InterfaceClassShareMethod();
			icm.fight();
			icm.bright();
			icm.tight();
		}
}
Output:
InterfaceClassShareMethod1 class
this is bright
this is tight
```

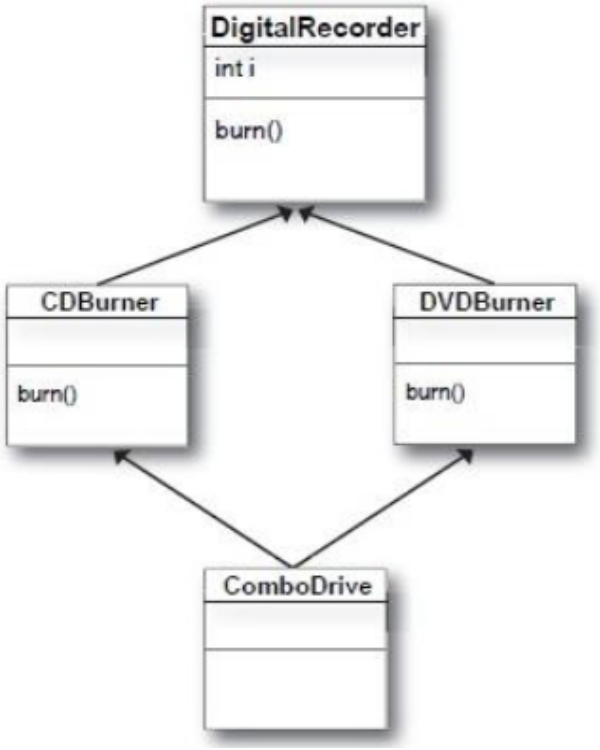
Note : Even comments for ‘abstract’ is removed it will work fine .

➔ If in the previous case, if common method will be implemented in the subclass then it will **overloads** the method of superclass.

3. <u>Multiple inheritance in java</u> :

➔ Multiple inheritance in java will be done through interfaces. i.e. in java there no multiple inheritance through classes but through interfaces.

➔ Why the multiple inheritance is done through interfaces in java bcos of DDD (deadly diamond of death) problem in java. Example of DDD is shown below.



In the above diagram we can see that when an object of class ComboDrive is created and burn() is called on it then confusion will occur and it is called as DDD.

➔ But how if a class A want three different methods each from one of three class ?? How one will going to implement that since multiple inheritance is not available. Answer for this question is interface is not here for 'code reuse' , its only made for a particular type or context and upon that only everything works in java. And in almost all of the circumstances there are few chances that such scenario take place.

➔ What happens when a class extends a abstract class and implements an interface both having the exactly same abstract method.

```
package ch9Interfaces;

abstract class AbstrInterfaceClass1
{
	abstract void method1();
}

interface AbstrInterfaceClass2
{
	void method1();
}

class AbstrInterfaceClass3 extends AbstrInterfaceClass1 implements
AbstrInterfaceClass2

{
	public void method1()
	{
		System.out.println("inside method");
	}
}

public class AbstrInterfaceClass
{

	public static void main(String[] args)
	{
		AbstrInterfaceClass1 aic1 = new AbstrInterfaceClass3();
		AbstrInterfaceClass2 aic2 = new AbstrInterfaceClass3();

		aic1.method1();
		aic2.method1();
	}
}
Output:
inside method
inside method
```

➔ One of the advantages for interfaces(also for abstract classes) that although an interface object cannot be created but surely a reference can be created which can be used polymorphically. A simple example shown below.

```
package ch9Interfaces;

interface intfcone
	{
	void infmet1();
	}

class InterfacePolymorphicNature1 implements intfcone
	{
		public void infmet1()
		{
		System.out.println("inside class 1");
		}
	}
```

```
class InterfacePolymorphicNature2 implements intfcone
	{
		public void infmet1()
		{
			System.out.println("inside class 2");
		}
	}

class InterfacePolymorphicNature3 implements intfcone
	{
		public void infmet1()
		{
			System.out.println("inside class 3");
		}
	}

public class InterfacePolymorphicNature
	{
		/**
		 * polymorphic nature of interfaces
		 */
		public static void main(String[] args) {
			// TODO Auto-generated method stub
			intfcone inf1 = new InterfacePolymorphicNature1();
			inf1.infmet1();
			intfcone inf2 = new InterfacePolymorphicNature2();
			inf2.infmet1();
			intfcone inf3 = new InterfacePolymorphicNature3();
			inf3.infmet1();
		}
}
Output:
inside class 1
inside class 2
inside class 3
```

➔ Same above program can be rewritten using Strategy pattern as shown below.

```
package ch9Interfaces;

interface intfcone1
{
	void infmet1();
}

class InterPolStrategyPattern1 implements intfcone1
{
	public void infmet1()
	{
		System.out.println("inside class 1");
	}
}
class InterPolStrategyPattern2 implements intfcone1
{
	public void infmet1()
	{
		System.out.println("inside class 2");
```

```
	}
}

class InterPolStrategyPattern3 implements intfcone1
{
	public void infmet1()
	{
		System.out.println("inside class 3");
	}
}

class StrategyClass
{
	void strategyMethod(intfcone1 inf)
	{
		inf.infmet1();
	}
}

public class InterPolStrategyPattern
{
	/**
	 * polymorphic nature of interfaces using strategy pattern
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		InterPolStrategyPattern1 inf1 = new InterPolStrategyPattern1();
		InterPolStrategyPattern2 inf2 = new InterPolStrategyPattern2();
		InterPolStrategyPattern3 inf3 = new InterPolStrategyPattern3();
		StrategyClass strg = new StrategyClass();
		strg.strategyMethod(inf1);
		strg.strategyMethod(inf2);
		strg.strategyMethod(inf3);

	}
}
Output:
inside class 1
inside class 2
inside class 3
```

Generics – Java Tutorials

- Generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods.

- (GFAQ)Generics are basically invented to create generic collections type. Because all collections accepts objects of different type hence while retriving them one needs to casts all of them. Hence with Generics one can have homogenous collections.

- (GFAQ)With Generics one can create collections which are homogeneous in nature like LinkedList<String> and LinkedList<Integer> which accepts only String and Integer elements respectively.

- (GFAQ)Generic is useful because type mismatch could be found out during compile time as error rather than during runtime as exceptions.

```
LinkedList<String> ls = new LinkedList<String>();

list.add("abc");//fine

list.add(new Date());//error
```

Without Generics above would be

```
LinkedList ls = new LinkedList();

list.add("abc");//fine

list.add(new Date());//fine
```

- Generics notations:

T – used to denote type
E – used to denote element
K – keys
V - values
N – for numbers

- Advantages of using Generics

  - Stronger type checks at compile time.

    Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

  - Elimination of casts.
    The following code snippet without generics requires casting:

```
List list = new ArrayList();
list.add("hello");
```

```
String s = (String) list.get(0);
```

When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

- Enables programmers to implement generic algorithms.

  By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

- Generic Types

  Generic type is generic class or interface which is parameterized over types.

  Example:

  Lets consider a non-generic class Box. It has two methods set() to put something inside the box and get() to retrieve it back.

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

  We can see that Box accepts and returns an 'object'. One is free to pass in whatever he wants. One part of code may place an 'Integer' and expect 'Integer' back, while other part of code may pass a 'String' resulting in a runtime error.

  Generic Class is defined as class name<T1, T2,....Tn> {/* . . . */} here T1, T2,...Tn are type parameters which could be used anywhere inside the class.

```
public class Box<T>
{
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

  A type variable can be any **non-primitive** type you specify: any class type, any interface type, any array type, or even another type variable. All types are allowed except enum types, anonymous inner classes and exception classes.

  Example:

```
package javaTutorial;

class SimpleGeneric1<T>
{
```

```
        T t;
        void setType(T t1)
        {
                this.t = t1;
                System.out.println(t);
        }
        T getType()
        {
                return t;
        }
}

public class SimpleGeneric {
        public static void main(String[] args)
        {
                SimpleGeneric1<String> in = new SimpleGeneric1<String>();
                in.setType("jayant");
                System.out.println(in.getType());
        }
    }
    Output:
    jayant
    jayant
```

- Type parameter naming convention

  By convention, type parameter names are single, uppercase letters. Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

  The most commonly used type parameter names are.

  E - Element (used extensively by the Java Collections Framework)

  K - Key

  N - Number

  T - Type

  V - Value

  S,U,V etc. - 2nd, 3rd, 4th types

- Invoking and Instantiating a Generic Type

  To reference the generic box class, one must perform 'generic type invocation' which is done by replacing type 'T' with some concrete value.

  ```
  Ex. Box<Integer> integerBox;
  ```

  It is something like method invocation, instead of passing an argument to a method here we pass 'type argument' – Integer this case.

Note: There is difference between ‘type parameter’ and ‘type argument’. In above example ‘T’ is type parameter and ‘Integer’ is type arguement.

To instantiate Box class we should do

```
Box<Integer> integerBox = new Box<Integer>();
```

From Java 7, ‘the diamond’ has been introduced, so now instance of Box can be created as

```
Box<Integer> integerBox = new Box<>();
```

- Multiple type parameter

A generic class can have multiple type parameters too.

Example:

```
package javaTutorial;

class MultipleTypeParam1<K, V>
{
   K key;
   V value;
   MultipleTypeParam1(K key1, V value1)
   {
         this.key = key1;
         this.value = value1;
   }
   K getK()
   {
         return key;
   }
   V getV()
   {
         return value;
   }
}
public class MultipleTypeParam
{
   public static void main(String[] args)
   {
         MultipleTypeParam1<Integer, String> mtp1 = new
MultipleTypeParam1<Integer, String>(5, "odd");
         MultipleTypeParam1<String, String> mtp2 = new
MultipleTypeParam1<String, String>("Jayant", "Joshi");
         System.out.println(mtp1.getK());
         System.out.println(mtp1.getV());
         System.out.println(mtp2.getK());
         System.out.println(mtp2.getV());
   }
```

```
}
Output:
5
odd
Jayant
Joshi
```

- Parameterized types

- Raw Types
  - A *raw type* is the name of a generic class or interface without any type arguments. For example, given the genericBox class:

    ```
    public class Box<T> {
        public void set(T t) { /* ... */ }
        // ...
    }
    ```

    To create a parameterized type of Box<T>, you supply an actual type argument for the formal type parameter T:

    ```
    Box<Integer> intBox = new Box<>();
    ```

    If the actual type argument is omitted, you create a raw type of Box<T>:

    ```
    Box rawBox = new Box();
    ```

    Therefore, Box is the raw type of the generic type Box<T>. However, a non-generic class or interface type is *not* a raw type.

  - (GFAQ)Raw types have been permitted to facilitate interfacing with non-generic (legacy) code.
  - For backward compatibility, assigning a parameterized type to its raw type is allowed:

    ```
    Box<String> stringBox = new Box<>();
    Box rawBox = stringBox;               // OK
    ```

    But if you assign a raw type to a parameterized type, you get a warning:

    ```
    Box rawBox = new Box();           // rawBox is a raw type of Box<T>
    Box<Integer> intBox = rawBox;     // warning: unchecked conversion
    ```

  - You also get a warning if you use a raw type to invoke generic methods defined in the corresponding generic type:

    ```
    Box<String> stringBox = new Box<>();
    Box rawBox = stringBox;
    rawBox.set(8);  // warning: unchecked invocation to set(T)
    ```

    Example:

    ```
    package javaTutorial;

    class RawTypesClass1<R>
    {
          R r;
    ```

```
	void SetRaw(R r1)
	{
		r = r1;
	}
	void PrintRaw()
	{
		System.out.println(r);
	}
}

public class RawTypesClass {

	public static void main(String[] args)
	{
		RawTypesClass1<Integer> rt1 = new RawTypesClass1<Integer>();
		rt1.SetRaw(5);
		rt1.PrintRaw();

//Warning:RawTypesClass1 is a raw type. References to generic type
RawTypesClass1<R> should be parameterized


	/*warning on this line*/RawTypesClass1 rt2 = new RawTypesClass1();
	/*warning*/rt2.SetRaw(10);
		rt2.PrintRaw();
		rt2 = rt1;// assigning parameterized type to its raw type is
allowed
		rt2.PrintRaw();
		/*warning*/rt1 = rt2;//reverse generate a warning
		rt1.PrintRaw();
	}
}
Output:
5
10
5
5
```

- Generic methods

  - Generic methods are the one which introduces their own generic type but the scope of type parameter is limited to scope.
  - Syntax:
    The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.
  - Example:

```
package javaTutorial;

class GenericMethClass1<T>
{
   T t;
   GenericMethClass1(T t1)
   {
```

```
            t = t1;
    }
    <I> void meth1(I i1)
    {
            if(i1 == t)
            {
                    System.out.println("success");
            }
            if(i1 != t)
            {
                    System.out.println("failure");
            }
    }
}

public class GenericMethClass
{
    public static void main(String[] args)
    {
            GenericMethClass1<Integer> gm = new
GenericMethClass1<Integer>(10);
            gm.<Integer>meth1(10);
            //gm.<String>meth1(10);//The parameterized method
<String>meth1(String) of type GenericMethClass1<Integer> is not
applicable for the arguments (Integer)
            //gm.<Integer>meth1("Jayant");//The parameterized method
<Integer>meth1(Integer) of type GenericMethClass1<Integer> is not
applicable for the arguments (String)
            GenericMethClass1<String> gm1 = new
GenericMethClass1<String>("String");
            gm1.<String>meth1("Jayant");
    }
}
Output:
success
failure
```

Example 2:

```
package generics;

class GenericMeth1<T>
{
    T t;
    GenericMeth1(T t1)
    {
            t = t1;
    }
    <M extends String> void genMethod(M m)
    {
            if (m == t)
            {
                    System.out.println("success");
            }
            else
```

```
            if (m != t)
            {
                  System.out.println("failure");
            }

      }
}

public class GenericMeth
{
      public static void main(String[] args)
      {
            GenericMeth1<String> gm11 = new GenericMeth1<>("jayant");
            GenericMeth1<Integer> gm12 = new GenericMeth1<>(10);

            gm11.<String>genMethod("jayant");
            gm12.<String>genMethod("joshi");

      }
}
Output:
success
failure
```

- Bounded Type Parameters

  - Sometimes there could be times when one needs to restrict type argument in parameterized types. In the previous program we have seen that method 'meth1' has been used for both <Integer> and <String> type arguments. But if one wanted to restrict them to particular type argument then java provide 'bounded type parameters'.
  - To declare bound type parameter, list the type parameter's name, followed by extends keyword, followed by upper bound. Example is shown below.

```
package javaTutorial;

class BoundedTypeParameters1<I extends Number>
{
      I i1;
      void set(I i2)
      {
            i1 = i2;
      }
      void printvalue()
      {
            System.out.println("value is "+i1);
      }
}

public class BoundedTypeParameters
{
      public static void main(String[] args)
      {
            BoundedTypeParameters1<Integer> bt = new
            BoundedTypeParameters1<Integer>();
            bt.set(5);
```

```
            bt.printvalue();
            //BoundedTypeParameters1<String> bt1 = new
   BoundedTypeParameters1<String>();
            //Error:Bound mismatch: The type String is not a valid
   substitute for the bounded parameter <I extends Number> of the type
   BoundedTypeParameters1<I>
      }
   }
   Output:
   value is 5
```

- We can have classes and Interfaces too as bounded types.
  - When we have a class statement as Class A <T extends SimpleClass> it means that T can only be SimpleClass or its subclasses.
  - Example:

```
package javaTutorial;

class AppleClass<C>
{
      C color1;
      AppleClass(C color)
      {
            color1 = color;
      }
      void setColor(C color)
      {
            color1 = color1;
      }
      C getColor()
      {
            return color1;
      }
}

class RedAppleContainer<A extends AppleClass>
{
      A apple;
      void checkApple(A apple1)
      {
            if(apple1.color1 == "red")
            {
                  apple = apple1;
                  System.out.println("Its a red apple");
            }
            else
            {
                  System.out.println("Its not a red apple its
                  "+apple1.color1);
            }
      }
}

class SmallApple<S> extends AppleClass<S>
{
      S color2;
```

```
	SmallApple(S color)
	{
		super(color);
		color2 = color;
	}
	S getColor1()
	{
		return color2;
	}
}

public class AppleContainerClass {

	public static void main(String[] args)
	{
		AppleClass<String> apple1 = new AppleClass<String>("red");
		System.out.println("color of apple is "+apple1.getColor());

		AppleClass<String> apple2 = new AppleClass<String>("green");
		System.out.println("color of apple is "+apple2.getColor());

		SmallApple<String> sa = new SmallApple<String>("red");
		System.out.println("color of small apple is
"+sa.getColor1());

		SmallApple<String> sa2 = new SmallApple<String>("green");
		System.out.println("color of small apple is
"+sa2.getColor1());


		RedAppleContainer<AppleClass<String>> rac = new
		RedAppleContainer<AppleClass<String>>();
		rac.checkApple(apple1);
		rac.checkApple(apple2);
		rac.checkApple(sa);
		rac.checkApple(sa2);

		RedAppleContainer<SmallApple<String>> sac = new
		RedAppleContainer<SmallApple<String>>();
		sac.checkApple(sa);
	}
}
Output:
color of apple is red
color of apple is green
color of small apple is red
color of small apple is green
Its a red apple
Its not a red apple its green
Its a red apple
Its not a red apple its green
Its a red apple
```

- Generic Method:
  The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.

```
package javaTutorial;

class Pair<K, V>
{
   K key;
   V value;
   public Pair(K k1, V v1)
   {
         this.key = k1;
         this.value = v1;
   }
   public void setKey(K key1)
   {
         this.key = key1;
   }
   public void setValue(V value1)
   {
         this.value = value1;
   }
   public K getKey()
   {
         return key;
   }
   public V getValue()
   {
         return value;
   }
}

class Util
{
   //generic method
   public static <K, V> boolean compare(Pair<K,V> p1, Pair<K, V> p2)
   {
          return p1.getKey().equals(p2.getKey()) &&
         p1.getValue().equals(p2.getValue());
   }
}

public class GenericMethodClass
{
   public static void main(String[] args)
   {
         Pair<Integer, String> p1 = new Pair<Integer, String>(1, "apple");
         Pair<Integer, String> p2 = new Pair<Integer, String>(2, "pear");
         Pair<Integer, String> p3 = new Pair<Integer, String>(1, "apple");
         boolean same1 = Util.compare(p1, p2);
         boolean same2 = Util.compare(p1, p3);
         System.out.println(same1);
         System.out.println(same2);
   }
```

```
    }
    Output
    false
    true
```

Example 2

```
package javaTutorial;

public class GenericMethodTest
{
   // generic method printArray
   public static < E > void printArray( E[] inputArray )
   {
      // Display array elements
         for ( E element : inputArray ){
            System.out.printf("%s ", element); //its printf
         }
         System.out.println();
    }

    public static void main( String args[] )
    {
        // Create arrays of Integer, Double and Character
        Integer[] intArray = { 1, 2, 3, 4, 5 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

        System.out.println( "Array integerArray contains:" );
        printArray( intArray  ); // pass an Integer array

        System.out.println( "\nArray doubleArray contains:" );
        printArray( doubleArray ); // pass a Double array

        System.out.println( "\nArray characterArray contains:" );
        printArray( charArray ); // pass a Character array
    }
   }
   Output
   Array integerArray contains:
   1 2 3 4 5

   Array doubleArray contains:
   1.1 2.2 3.3 4.4

   Array characterArray contains:
   H E L L O
```

- Like raw types we can call generic methods also as shown below

```
package generics;

class GenMethCheck1
{
      <T> void method()
      {
```

```
            System.out.println("Inside method");
      }
}

public class GenMethCheck
{
      public static void main(String[] args)
      {
            GenMethCheck1 gmc = new GenMethCheck1();
            gmc.method();
      }
}
Output:
Inside method
```

- Generic Interface:
  - Example

```
package javaTutorial;

      interface StringInterface<S extends String>
      {
            void printString(S s);
      }

      class GenStringClass<S extends String> implements
      StringInterface<String>
      {
            public void printString(String str)
            {
                  System.out.println(str);
            }
      }

      public class GenStringInterfaceClass
      {
            public static void main(String[] args)
            {
                  GenStringClass<String> gsc1 = new GenStringClass<String>();
                  gsc1.printString("java");
                  gsc1.printString("generics");
                  //GenStringClass<Integer> gsc2 = new
      GenStringClass<Integer>();//bound mismatch Integer
            }
      }
      Output:
      java
      generics
```

- Comparable Interface:

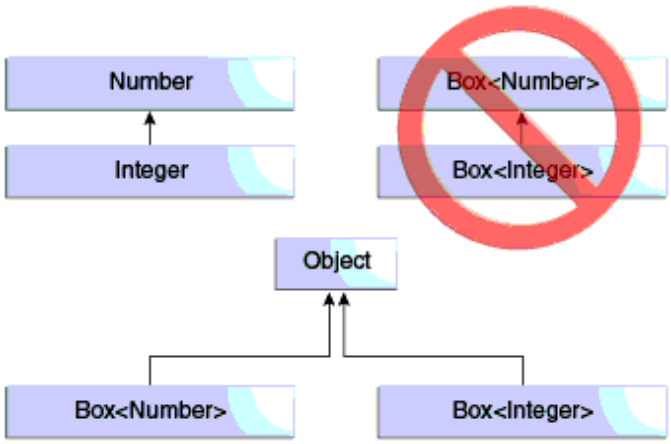
```
package javaTutorial;

class CompareToClass<T extends String> implements Comparable<T>
{
```

```
    T t;
    CompareToClass(T t1)
    {
          t = t1;
    }
    public int compareTo(T t2)
    {
          int l = t.length() - t2.length();
          return l;
    }
}

public class GenCompareToInterfaceClass
{
    public static void main(String[] args)
    {
          CompareToClass<String> ctc = new
CompareToClass<String>("Generic");
          System.out.println(ctc.compareTo("java"));
    }
}
Output:
3
```

- Multiple Bounds:
- Suppose we have a class

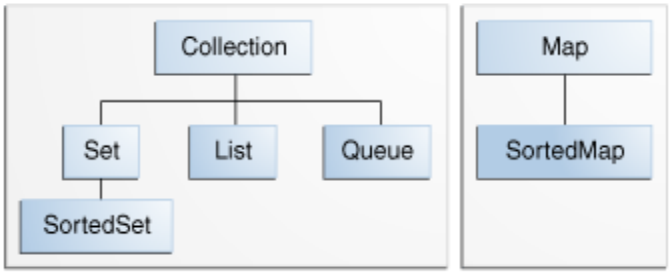
  `public void boxTest(Box<Number> n) { /* ... */ }`

  One can wonder what is the argument boxTest() will accept ? It may seem that Box<Number> could accept Box<Integer> or Box<Double> but its not correct. Given two concrete types A and B (for example, Number and Integer), MyClass<A> has no relationship to MyClass<B>, regardless of whether or not A and B are related. The common parent of MyClass<A> and MyClass<B> is Object.



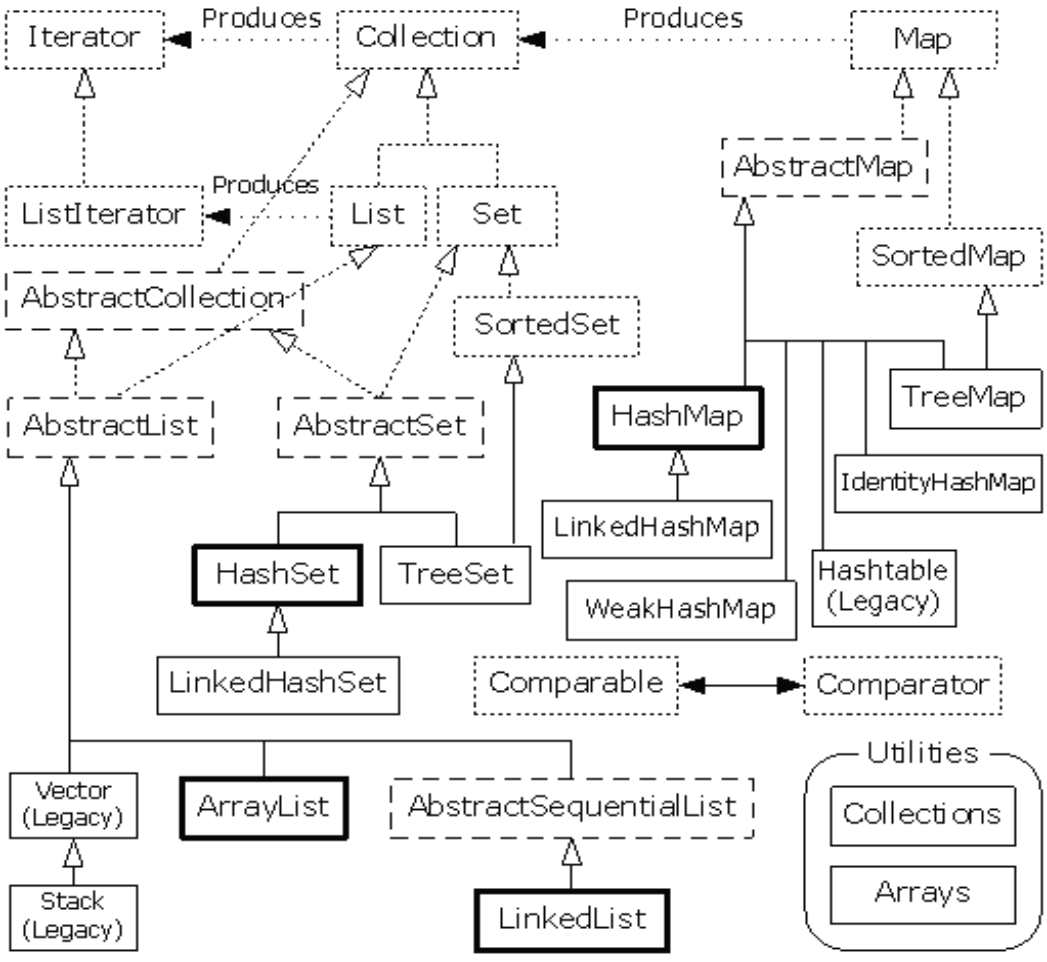
Collections

1. Core collections interfaces are



Here Set is special kind of collections and SortedSet is special kind of Set..it goes like that.

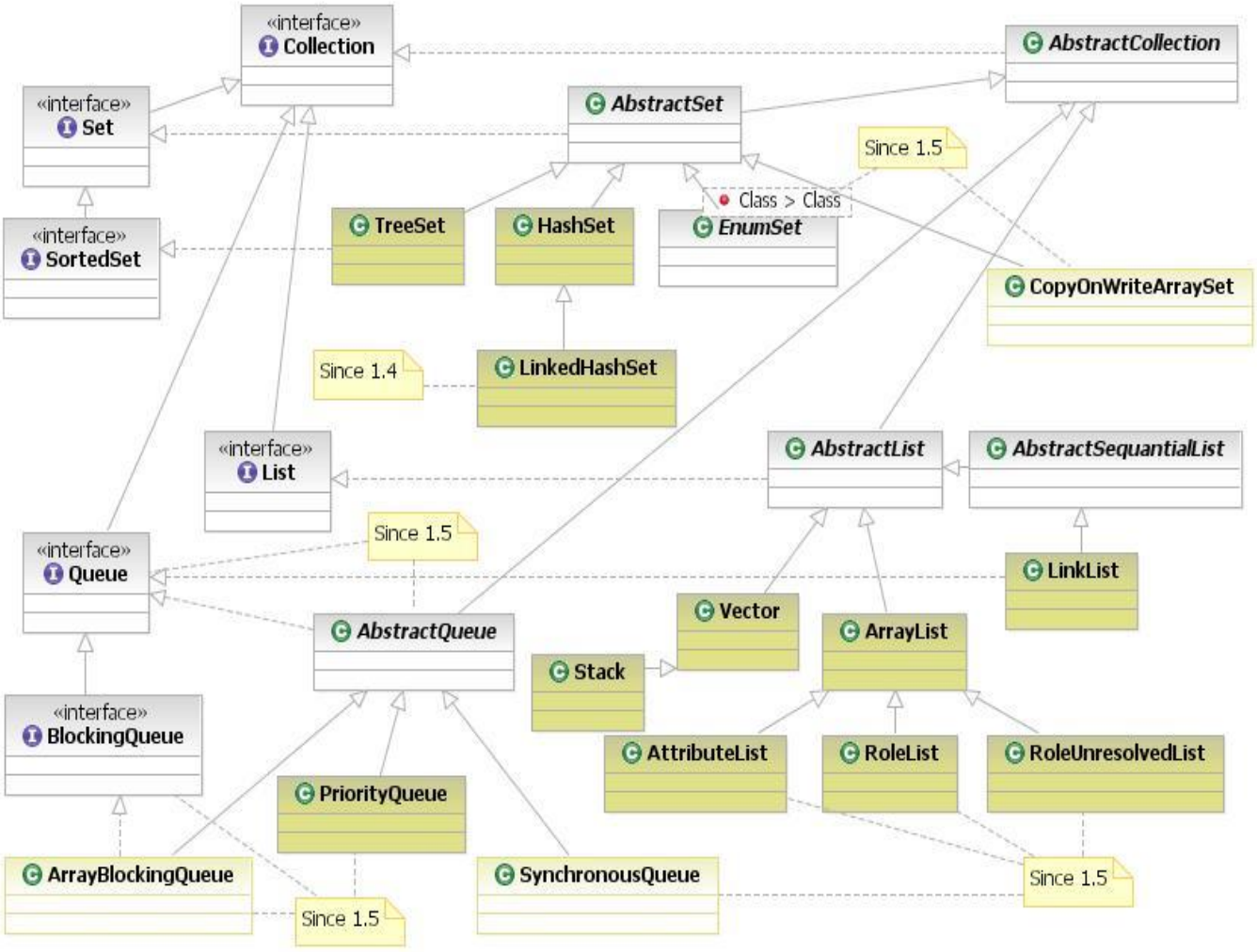
2. Elaborated Collection framework(without queue: Thinking In java 4th edition)



3. Also there are two distinct tree as show in diagram above. The Map is not a true collection.
4. All collections are generic. For example, declaration for collection interface is

```
public interface Collection<E>...
```

- One more collection diagram(including queue).



<---------- Implements

<---------- Extends

5. While declaring a collection instance, one should specify the type of object contained in it. Specifying the type allows compiler to verify the correct type of the object thus reducing errors during runtime.
6. Collection: Collection is the root of collection hierarchy. This is the interface which all collections implement. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific sub-interfaces, such as `Set` and `List`.

   Set: Is the collection which cannot contain duplicate elements. This interface models the mathematical set and is used to represent sets, such as cards comprising a poker hand.

   List: An ordered collection (Sometimes called a sequence). List can contain duplicate elements. A user of the List has precise control over it that where is each element has been inserted and can access them using their integer index (position).

   Queue: A collection which is used to hold multiple elements prior to processing. Besides basic collection operations, a queue provides additional insertion, extraction and inspection operations. Queues typically, but not necessarily, order elements in FIFO (first in, first out) manner. Every queue implementation must specify its ordering properties.

   Map: An object that maps keys to values. A map cannot contain duplicate keys and each key at most map to one value.

   SortedSet: A set that maintains its elements in ascending order. Several other operations have been provided to take the advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.

   SortedMap: A map that maintains its ordering in ascending key order. They are used for natural ordered collections of key/value pairs, such as dictionaries and telephone directories.
7. Also we can see in above diagram an abstract class AbstractCollection extends Collection interface. AbstractCollection provides skeleton implementation of Collection interface just to minimize the effort needed to implement it.
8. Then other abstract classes like AbstractSet, AbstractList etc extends AbstractCollection and implements interfaces like Set, List etc. Then from this AbstractSet, AbstractList etc we will going to create our HashSet and Arraylist concrete classes.
9. Optional Operations:

   - To keep the core interfaces manageable , the java platform doesn't provide separate interface for each variant of each collection type(such variants might include immutable, fixed size and append only) . Instead, the modification operation in each interface is designated **optional**.
     Example : For collection interface method add(E e) is marked as optional as shown below.

Modifier and Type	Method and Description
boolean	**add**(**E** e) Ensures that this collection contains the specified element (optional operation).
boolean	**addAll**(**Collection**<? extends **E**> c) Adds all of the elements in the specified collection to this collection (optional operation).

- A given implementation may elect not to support all operations. If unsupported operation is invoked, a collection throws `UnsupportedOperationException`. These optional operations will be already defined as unsupported by abstract classes like AbstractCollection, AbstractList etc as shown below.

  Example 1: add(E paramE) method of Collection interface is defined in AbstractCollection class as

```
public boolean add(E paramE)
  {
      throw new UnsupportedOperationException();
  }
```

  Example 2 : Method add(int paramInt, E paramE) of the interface List is defined in AbstractList class as

```
public void add(int paramInt, E paramE)
  {
      throw new UnsupportedOperationException();
   }
```

- i.e. Now its completely left to implementation to define(override) these operations in their implementation if they want to support them. All java platform's general purpose implementation (As shown in table below under implementations) supports all optional (i.e. define) operations. Example class ArrayList defines (overrides) both `add(E paramE)` and `add(int paramInt, E paramE)` methods .

10. Collection Interface:

- Any class which defines a collection should implement this interface.
- Collection is a generic interface with declaration
- interface Collection<E>

Method	Description
boolean add(E *obj*)	Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Returns **false** if *obj* is already a member of the collection and the collection does not allow duplicates.
boolean addAll(Collection<? extends E> *c*)	Adds all the elements of *c* to the invoking collection. Returns **true** if the collection changed (i.e., the elements were added). Otherwise, returns **false**.
void clear( )	Removes all elements from the invoking collection.
boolean contains(Object *obj*)	Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**.
boolean containsAll(Collection<?> *c*)	Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**.
boolean equals(Object *obj*)	Returns **true** if the invoking collection and *obj* are equal. Otherwise, returns **false**.
int hashCode( )	Returns the hash code for the invoking collection.
boolean isEmpty( )	Returns **true** if the invoking collection is empty. Otherwise, returns **false**.
Iterator<E> iterator( )	Returns an iterator for the invoking collection.
boolean remove(Object *obj*)	Removes one instance of *obj* from the invoking collection. Returns **true** if the element was removed. Otherwise, returns **false**.
boolean removeAll(Collection<?> *c*)	Removes all elements of *c* from the invoking collection. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
boolean retainAll(Collection<?> *c*)	Removes all elements from the invoking collection except those in *c*. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
int size( )	Returns the number of elements held in the invoking collection.
Object[ ] toArray( )	Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
<T> T[ ] toArray(T *array*[ ])	Returns an array that contains the elements of the invoking collection. The array elements are copies of the collection elements. If the size of *array* equals the number of elements, these are returned in *array*. If the size of *array* is less than the number of elements, a new array of the necessary size is allocated and returned. If the size of *array* is greater than the number of elements, the array element following the last collection element is set to **null**. An **ArrayStoreException** is thrown if any collection element has a type that is not a subtype of *array*.

- The collection methods can throw following exceptions

i. **UnsupportedOperationException**: When collection cannot be modified.

ii. **ClassCastException**: When attempt is made to add incompatible object to the collection.

iii. **NullPointerException**: When attempt is made to store a null object.

iv. **IllegalArgumentException**: If an invalid argument is used.

v. **IllegalStateException**: If attempt is made to add an element to a fixed length collection which is full.

vi. **ArrayStoreException** : If attempt is made to return an array contains collection elements and type mismatch occurs.

- Traversing the collection

There are two ways to traverse through the collection

i. With for-each

```
for (Object o : collection)
System.out.println(o);
```

ii. Using Iterator

One can get the iterator of a collection by calling iterator method. Following is the Iterator interface.

```
public interface Iterator<E>
{
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

hasNext() – returns true If iterator has more elements

next() – returns next element in the iteration.

remove() – removes the last element that was returned by next

11. Iterator and remove() method :

- The method remove() of interface Iterator removes the element from the collection. But one should call next() Method before calling remove second time else `IllegalStateException` will be thrown. Example for the same is shown below.

```
package collections;

import java.util.*;

public class IteratorRemoveMethod
{
      public static void main(String[] args)
      {
            ArrayList<String> al = new ArrayList<String>();
            al.add("one");
            al.add("two");
            al.add("three");
            System.out.println("al after addn "+al);

            Iterator<String> it = al.iterator();
            while(it.hasNext())
            {
```

```
                    String str = it.next();
                    it.remove();
                    it.remove();
                }
            }
        }
```

Output:

```
al after addn [one, two, three]
Thread [main] (Suspended (exception IllegalStateException))
                ArrayList$Itr.remove() line: 804
                IteratorRemoveMethod.main(String[]) line: 20
```

12. Set Interface:

- It contains methods inherited from *Collection* and adds the restriction that duplicate elements are Prohibited, therefore add() method returns false if attempt is made to add duplicate element to set.
- Set doesn't define any additional method of its own.
- Syntax : `public interface Set<E> extends Collection<E>`

13. SortedSet Interface :

- SortedSet interface extends Set and declares the behaviour of a set sorted in ascending order.
- Syntax : `public interface SortedSet<E> extends Set<E>`
- In addition to method of Set, SortedSet declares some extra methods as listed below.

Method	Description
Comparator<? super E> comparator( )	Returns the invoking sorted set's comparator. If the natural ordering is used for this set, **null** is returned.
E first( )	Returns the first element in the invoking sorted set.
SortedSet<E> headSet(E *end*)	Returns a **SortedSet** containing those elements less than *end* that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.
E last( )	Returns the last element in the invoking sorted set.
SortedSet<E> subSet(E *start*, E *end*)	Returns a **SortedSet** that includes those elements between *start* and *end*–1. Elements in the returned collection are also referenced by the invoking object.
SortedSet<E> tailSet(E *start*)	Returns a **SortedSet** that contains those elements greater than or equal to *start* that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

14. NavigableSet:

- NavigableSet interface extends SortedSet and declares the behaviour of a collection that supports the retrieval of elements based on the closest match to a given value or values.
- Syntax : `public interface NavigableSet<E> extends SortedSet<E>`

Method	Description
E ceiling(E *obj*)	Searches the set for the smallest element *e* such that *e* >= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
Iterator<E> descendingIterator( )	Returns an iterator that moves from the greatest to least. In other words, it returns a reverse iterator.
NavigableSet<E> descendingSet( )	Returns a **NavigableSet** that is the reverse of the invoking set. The resulting set is backed by the invoking set.
E floor(E *obj*)	Searches the set for the largest element *e* such that *e* <= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
NavigableSet<E> headSet(E *upperBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set.
E higher(E *obj*)	Searches the set for the largest element *e* such that *e* > *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
E lower(E *obj*)	Searches the set for the largest element *e* such that *e* < *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
E pollFirst( )	Returns the first element, removing the element in the process. Because the set is sorted, this is the element with the least value. **null** is returned if the set is empty.
E pollLast( )	Returns the last element, removing the element in the process. Because the set is sorted, this is the element with the greatest value. **null** is returned if the set is empty.
NavigableSet<E> subSet(E *lowerBound*, boolean *lowIncl*, E *upperBound*, boolean *highIncl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound* and less than *upperBound*. If *lowIncl* is **true**, then an element equal to *lowerBound* is included. If *highIncl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set.
NavigableSet<E> tailSet(E *lowerBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound*. If *incl* is **true**, then an element equal to *lowerBound* is included. The resulting set is backed by the invoking set.

Note: In above table method higher(E obj) returns the least element in this set strictly greater than the given element, or null if there is no such element.

15. List Interface:

- List interface extends collection and declares the behaviour of a collection that stores a sequence of elements(ordered collection also known as a ***sequence***).

- Apart from methods inherited through Collection list have following additional methods
- List also provides its own richer iterator called ' `ListIterator`'. ListIterator interface has following methods

Method	Description
void add(E *obj*)	Inserts *obj* into the list in front of the element that will be returned by the next call to **next( )**.
boolean hasNext( )	Returns **true** if there is a next element. Otherwise, returns **false**.
boolean hasPrevious( )	Returns **true** if there is a previous element. Otherwise, returns **false**.
E next( )	Returns the next element. A **NoSuchElementException** is thrown if there is not a next element.
int nextIndex( )	Returns the index of the next element. If there is not a next element, returns the size of the list.
E previous( )	Returns the previous element. A **NoSuchElementException** is thrown if there is not a previous element.
int previousIndex( )	Returns the index of the previous element. If there is not a previous element, returns –1.
void remove( )	Removes the current element from the list. An **IllegalStateException** is thrown if **remove( )** is called before **next( )** or **previous( )** is invoked.
void set(E *obj*)	Assigns *obj* to the current element. This is the element last returned by a call to either **next( )** or **previous( )**.

- Syntax : public interface **List<E>** extends Collection<E>

Method	Description
void add(int *index*, E *obj*)	Inserts *obj* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
boolean addAll(int *index*, Collection<? extends E> *c*)	Inserts all elements of *c* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns **true** if the invoking list changes and returns **false** otherwise.
E get(int *index*)	Returns the object stored at the specified index within the invoking collection.
int indexOf(Object *obj*)	Returns the index of the first instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.

int lastIndexOf(Object *obj*)	Returns the index of the last instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
ListIterator<E> listIterator( )	Returns an iterator to the start of the invoking list.
ListIterator<E> listIterator(int *index*)	Returns an iterator to the invoking list that begins at the specified *index*.
E remove(int *index*)	Removes the element at position *index* from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
E set(int *index*, E *obj*)	Assigns *obj* to the location specified by *index* within the invoking list. Returns the old value.
List<E> subList(int *start*, int *end*)	Returns a list that includes elements from *start* to *end*–1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

16. Queue Interface:

- A collection which is used to hold multiple elements prior to processing. Besides basic collection operations, a queue provides additional insertion, extraction and inspection operations. Queues typically, but not necessarily, order elements in FIFO (first in, first out) manner. Every queue implementation must specify its ordering properties.
- Syntax : `public interface Queue<E> extends Collection<E>`
-

Method	Description
E element( )	Returns the element at the head of the queue. The element is not removed. It throws **NoSuchElementException** if the queue is empty.
boolean offer(E *obj*)	Attempts to add *obj* to the queue. Returns **true** if *obj* was added and **false** otherwise.
E peek( )	Returns the element at the head of the queue. It returns **null** if the queue is empty. The element is not removed.
E poll( )	Returns the element at the head of the queue, removing the element in the process. It returns **null** if the queue is empty.
E remove( )	Removes the element at the head of the queue, returning the element in the process. It throws **NoSuchElementException** if the queue is empty.

17. Deque Interface:

- The **Deque** interface extends **Queue** and declares the behavior of a double-ended queue. Double-ended queues can function as standard, first-in, first-out queues or as last-in, first-out stacks.
- public interface **Deque<E>** extends Queue<E>

Method	Description
void addFirst(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
void addLast(E *obj*)	Adds *obj* to the tail of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
Iterator<E> descendingIterator( )	Returns an iterator that moves from the tail to the head of the deque. In other words, it returns a reverse iterator.
E getFirst( )	Returns the first element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty.
E getLast( )	Returns the last element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty.
boolean offerFirst(E *obj*)	Attempts to add *obj* to the head of the deque. Returns **true** if *obj* was added and **false** otherwise. Therefore, this method returns **false** when an attempt is made to add *obj* to a full, capacity-restricted deque.
boolean offerLast(E *obj*)	Attempts to add *obj* to the tail of the deque. Returns **true** if *obj* was added and **false** otherwise.
E peekFirst( )	Returns the element at the head of the deque. It returns **null** if the deque is empty. The object is not removed.
E peekLast( )	Returns the element at the tail of the deque. It returns **null** if the deque is empty. The object is not removed.
E pollFirst( )	Returns the element at the head of the deque, removing the element in the process. It returns **null** if the deque is empty.
E pollLast( )	Returns the element at the tail of the deque, removing the element in the process. It returns **null** if the deque is empty.
E pop( )	Returns the element at the head of the deque, removing it in the process. It throws **NoSuchElementException** if the deque is empty.
void push(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
E removeFirst( )	Returns the element at the head of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
boolean removeFirstOccurrence(Object *obj*)	Removes the first occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*.
E removeLast( )	Returns the element at the tail of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
boolean removeLastOccurrence(Object *obj*)	Removes the last occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*.

18. Implementations:

General-purpose Implementations

Interfaces	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked list Implementations	Hash table + Linked list Implementations
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

19. Collection Classes:

Class	Description
AbstractCollection	Implements most of the **Collection** interface.
AbstractList	Extends **AbstractCollection** and implements most of the **List** interface.
AbstractQueue	Extends **AbstractCollection** and implements parts of the **Queue** interface.
AbstractSequentialList	Extends **AbstractList** for use by a collection that uses sequential rather than random access of its elements.
LinkedList	Implements a linked list by extending **AbstractSequentialList**.
ArrayList	Implements a dynamic array by extending **AbstractList**.
ArrayDeque	Implements a dynamic double-ended queue by extending **AbstractCollection** and implementing the **Deque** interface.
AbstractSet	Extends **AbstractCollection** and implements most of the **Set** interface.
EnumSet	Extends **AbstractSet** for use with **enum** elements.
HashSet	Extends **AbstractSet** for use with a hash table.
LinkedHashSet	Extends **HashSet** to allow insertion-order iterations.
PriorityQueue	Extends **AbstractQueue** to support a priority-based queue.
TreeSet	Implements a set stored in a tree. Extends **AbstractSet**.

20. ArrayList :

- Syntax :

```
public class ArrayList<E> extends AbstractList<E> implements List<E>,
RandomAccess, Cloneable, Serializable
```

- ArrayList class extends AbstractList and implement the list interface.
- ArrayList supports dynamic arrays that can grow as needed. In java we know that arrays are of fix size that means one must know in advance that how many elements an array will hold. But sometimes, you may not know till run time precisely large an array one need. To handle this situation, collections framework defines ArrayList.
- In essence ArrayList is a variable length array of object references.

- Dynamic array is also hold by legacy class Vector.
- ArrayLists are created with initial size, when the size is exceeded the collector will automatically enlarge.
- **ArrayList** has the constructors shown here:

`ArrayList()` Constructs an empty list with an initial capacity of ten.
`ArrayList(Collection<? extends E> c)` Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.
`ArrayList(int initialCapacity)` Constructs an empty list with the specified initial capacity.

-Example of ArrayList :

```
package collections;

import java.util.*;

public class ArrayListDemo
{
public static void main(String[] args)
   {
        ArrayList<String> al = new ArrayList<String>();
System.out.println("initial size of the arraylist is "+ al.size());

        //adding elements to arraylist al
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
        al.add(1, "A2");

        //size and contents after addition
        System.out.println("Size of arraylist after addition is "+
  al.size());
        System.out.println("content of al "+ al);
        System.out.println();

        //removing contents of al
        al.remove("F");
        al.remove(2);

        //size and contents after deletion
System.out.println("Size of arraylist after deletion is "+ al.size());
        System.out.println("content of al "+ al);
        System.out.println();
```

```
        //creating duplicate arraylist al1
        ArrayList<String> al1 = new ArrayList<String>();
        al1.addAll(al);
        System.out.println("content of al1 "+al1);
        System.out.println();

        //method contains()
        if(al.contains("C"))
        {
              System.out.println("contains");
        }
        else
        {
              System.out.println("dont contains");
        }
        System.out.println();

        //method equals()
        if(al.equals(al1))
        {
              System.out.println("al and al1 are equal");
        }
        else
        {
              System.out.println("al and al1 are unequal");
        }
        System.out.println();

        //hashcode for al
        System.out.println("hash code for al is '"+al.hashCode()+"'");
        System.out.println();

        //Using an iterator
        Iterator<String> it = al.iterator();
        System.out.println("contents of al through iterator:");
        while(it.hasNext())
        {
              String element = it.next();
              System.out.println(element+" ");
        }
        System.out.println();

        //equal works even after removing all elements
        al.removeAll(al);
        al1.removeAll(al1);
        System.out.println("content of al "+ al);
        System.out.println("content of al1 "+ al1);
        if(al.equals(al1))
        {
              System.out.println("al and al1 are equal");
        }
        else
        {
              System.out.println("al and al1 are unequal");
        }
        System.out.println();

        System.out.println(al.isEmpty());

    }
}
```

```
Output:
initial size of the arraylist is 0
Size of arraylist after addition is 7
content of al [C, A2, A, E, B, D, F]

Size of arraylist after deletion is 5
content of al [C, A2, E, B, D]

content of al1 [C, A2, E, B, D]

contains

al and al1 are equal

hash code for al is '152091896'

contents of al through iterator:
C
A2
E
B
D

content of al []
content of al1 []
al and al1 are equal

true
```

- Although the size of ArrayList increases automatically when more elements are added but one can increase the size manually using **ensureCapacity()** method. Signature of the method is void ensureCapacity(int *cap*). This will ensure minimum capacity not maximum limit i.e. if capacity is surpassed then it get increased automatically.
- Conversely if one wanted to reduce the capacity then we have **trimToSize()**. ArrayList backed by an array whose current capacity may be over hence capacity get increased automatically. Example let's say the initial capacity was 10 and if 11 elements are added then during addition of 11^{th} element it will automatically increases capacity to say 15. But as we know only 11 elements have been added the remaining memory of 4 elements just stay there until more elements got added. When one is sure of no more elements needs to be added then he can call **trimToSize()** method on that ArrayList which ensures that its capacity reduced to its current size i.e. 11 in this case.
- Obtaining an array from ArrayList:
    - Because of following (and other) reasons collection needs to be converted in arrays.
        1. To obtain faster processing time for certain operations.
        2. To pass an array to method which is not overloaded to accept a collection.
        3. To integrated collection based code with legacy code that doesn't understand collection.

(Note: If your List is fixed in size — that is, you'll never use remove, add, or any of the bulk operations other than containsAll — then the best option is to use Arrays.asList because arrays are not resizeable).

- There are two versions of toArray()

  object[ ] toArray( )
  <T> T[ ] toArray(T *array*[ ])

  The first returns array of Object and second one returns array of elements that has the same type as **T** which is more convenient.

- Example:

```
package collections;
import java.util.*;

public class ArrayListToArray
{
    public static void main(String[] args)
    {
          ArrayList<Integer> al = new ArrayList<Integer>();

          al.add(1);
          al.add(2);
          al.add(3);
          al.add(4);

          System.out.println("Contents of al : "+al);

          Integer ai[] = new Integer[al.size()];
          ai = al.toArray(ai);

          int sum = 0;

          for(int i : ai)
          sum += i;

          System.out.println("Sum is : "+sum);
    }
}
Output:
Contents of al : [1, 2, 3, 4]
Sum is : 10
```

- Arrays class has a static factory method called 'asList' which allows an array to be viewed as a List.
- <u>Difference between ArrayList and Vector</u>
  - All methods of Vector are synchronized but the methods of array list are not synchronized. Because of this Vector becomes slow and ArrayList is fast.
  - Vector and ArrayList both uses arrays internally as data structure. Therefore they are dynamically resizable, difference is in the way they internally resized. By default Vector doubles the size of its array when size is increased but ArrayList increases by half.

21. LinkedList Class:

- LinkedList Class extends **AbstractSequentialList** and implements **List**, **Queue** and **Deque** interfaces.
- `Syntax : public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, Serializable`
- A **LinkedList** is relatively slow for random access, but it has larger feature set than the **ArrayList**

- Constructor

LinkedList( )
LinkedList(Collection<? extends E> *c*)

The first constructor builds an empty linked list. The second constructor builds a linked list that is initialized with the elements of the collection *c*.

- Example 2:

```
package collections;

import java.util.LinkedList;

public class LinkedListDemo
{
   public static void main(String[] args)
   {
       LinkedList<String> ll = new LinkedList<String>();

       ll.add("F");
       ll.add("B");
       ll.add("D");
       ll.add("E");
       ll.add("C");
       ll.addLast("Z");
       ll.addFirst("A");
       ll.addFirst("A");

       ll.add(2, "A2");

       System.out.println("Original content of ll:"+ll);

       //search operation
       System.out.println("index of A2:"+ll.indexOf("A2"));
       System.out.println("index of A:"+ll.indexOf("A"));
       System.out.println("index of A:"+ll.lastIndexOf("A"));

       System.out.println("first element is "+ll.getFirst());
       System.out.println("last element is "+ll.getLast());

       //rang view operation
       System.out.println("sublist :"+ll.subList(0, 2));

       //LinkedList implements cloneable interface
```

```
        System.out.println("clone linkedlist is "+ll.clone());

        ll.remove("F");
        ll.remove(2);

        System.out.println("Content of ll after deletion:"+ll);

        ll.removeFirst();
        ll.removeLast();

        System.out.println("ll after removing first and last:"+ll);

        String val = ll.get(2);
        ll.set(2, val+" changed");

        System.out.println("Content of ll :"+ll);
    }
}
Output:
Original content of ll:[A, A, A2, F, B, D, E, C, Z]
index of A2:2
index of A:0
index of A:1
first element is A
last element is Z
sublist :[A, A]
clone linkedlist is [A, A, A2, F, B, D, E, C, Z]
Content of ll after deletion:[A, A, B, D, E, C, Z]
ll after removing first and last:[A, B, D, E, C]
Content of ll :[A, B, D changed, E, C]
```

- Above we can see, we have ll.clone() method also because LinkedList implements `Cloneable` interface which actually has no clone() method but interface is an indication for object.clone() method. Clone method here Returns a shallow copy of this LinkedList. (The elements themselves are not cloned).

22. Set Implementation:
    - Java has three general purpose Set implementations, they are HashSet, TreeSet and LinkedHashset.
    - HashSet which stores elements in hash table, is best performing implementation but it don't give any guarantee for order of iteration.
    - TreeSet which stores its elements in red-black tree, orders its elements based on their values. It is slower than HashSet.
    - LinkedHashSet which implements hash table with linked list running through it, orders its elements based on order they have been inserted into it(insertion order).

    Example of different flavours of sets :

```
package collections;

import java.util.*;

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

```
        ArrayList<String> Al = new ArrayList<String>();
        Al.add("D");
        Al.add("X");
        Al.add("B");
        Al.add("C");
        Al.add("E");

        System.out.println("contents of Array List are :"+Al);

        HashSet<String> Hs = new HashSet<String>(Al);

        System.out.println("contents of Hash set are :"+Hs);

        LinkedHashSet<String> Lhs = new LinkedHashSet<String>(Al);
        System.out.println("contents of Linked Hash set are :"+Lhs);

        //if an element is re-inserted then insertion order is not
affected
        Lhs.add("Y");
        System.out.println("contents of Linked Hash set are :"+Lhs);

        TreeSet<String> ts = new TreeSet<String>(Lhs);
        System.out.println("contents of tree set are "+ts);
  }
  }
  Output:
  contents of Array List are :[D, X, B, C, E]
  contents of Hash set are :[D, E, B, C, X]
  contents of Linked Hash set are :[D, X, B, C, E]
  contents of Linked Hash set are :[D, X, B, C, E, Y]
  contents of tree set are [B, C, D, E, X, Y]
```

Above we can see that how ArrayList allows duplicate while Hash set don't accept duplicates. Next it can be seen that Linked Hash Set maintains insertion order(not natural ordering) and put D in the last place. And finally tree set copies same elements of linked hash set but maintains natural ordering of its elements.

23. HashSet Class:

- Syntax : `public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, java.io.Serializable`
- HashMap is backed by HashTable (Actually a HashMap instance, if one will open the HashSet source code file then find the use of an HashMap instance and HashMap has same code as HashTable only difference is that HashMap methods are not synchronized and permits null values). Similarly HashSet methods calls corresponding HashMap methods using HashMap instance. Example

```
public int size()
  {
      return map.size();
  }
```

- HashSet doesn't define any additional methods beyond those provided by its super classes and interfaces.

- Constructors:

Constructors	
Constructor and Description	
HashSet()	Constructs a new, empty set; the backing HashMap instance has default initial capacity (16) and load factor (0.75).
HashSet(**Collection**<? extends **E**> c)	Constructs a new set containing the elements in the specified collection.
HashSet(int initialCapacity)	Constructs a new, empty set; the backing HashMap instance has the specified initial capacity (default capacity is 16 and load factor 0.75).
HashSet(int initialCapacity, float loadFactor)	Constructs a new, empty set; the backing HashMap instance has the specified initial capacity and the specified load factor. The load factor should be between 0.0 to 1.0.

- Example of HashSet

```
package collections;

import java.util.*;

public class HashSetDemo
{
	public static void main(String[] args)
	{
		HashSet<Integer> hs = new HashSet<Integer>();
		hs.add(20);
		hs.add(10);
		hs.add(15);
		hs.add(15);
		hs.add(5);

		System.out.println("hs after additions "+hs);

		hs.remove(5);
		System.out.println("hs after deletion "+hs);

	System.out.println("hs contains object? "+hs.contains(15));
	System.out.println("hs is empty? "+hs.isEmpty());

	HashSet<Integer> hs1 = new HashSet<Integer>();
	hs1.add(2);
	hs1.add(4);
```

```
            hs1.add(10);

            HashSet<Integer> hs2 = new HashSet<Integer>();
            hs2.add(15);

            System.out.println("hs1 after addtn "+hs1);
            System.out.println("hs2 after addtn "+hs2);

            //equal - should be exactly same
            System.out.println("hs equal hs1 ?"+hs.equals(hs1));
            System.out.println("hs equal hs2 ?"+hs.equals(hs2));

            //containsAll
            System.out.println("hs contains all of hs1?
"+hs.containsAll(hs1));
            System.out.println("hs contains all of hs2?
"+hs.containsAll(hs2));

            //removeAll - remove which is common and retain others
            hs.removeAll(hs1);
            System.out.println("hs after removeAll "+hs);

            //retainAll - retain which is common and remove others
            hs.add(10);
            hs.retainAll(hs1);
              System.out.println("hs after retainAll "+hs);
      }
}
Output:
```

```
hs after additions [5, 20, 10, 15]
hs after deletion [20, 10, 15]
hs contains object? true
hs is empty? false
hs1 after addtn [2, 4, 10]
hs2 after addtn [15]
hs equal hs1 ?false
hs equal hs2 ?false
hs contains all of hs1? false
hs contains all of hs2? true
hs after removeAll [20, 15]
hs after retainAll [10]
```

Above programme we can see the function equals only work for exactly equal hash sets. Function removeAll removes the common elements from the invoking set and retain other elements and the opposite is true for retainAll which retail all the common elements in invoking set and removes the others.

24. LinkedHashSet:

- `public class LinkedHashSet<E> extends HashSet<E> implements Set<E>, Cloneable, Serializable`
- It is hash table and linked list implementation of set interface.
- It extends HashSet and implements set. One can say LinkedHashSet is nothing but Hash set with linked list running through it as name suggest. i.e. **LinkedHashSet** maintains a linked list of

the entries in the set, in the order in which they were inserted. In another word LinkedHashSet is nothing but HashSet which maintains insertion order of the elements.

- Also it defines four constructors as shown below.

Constructors	
Constructor and Description	
`LinkedHashSet()`	Constructs a new, empty linked hash set with the default initial capacity (16) and load factor (0.75).
`LinkedHashSet(Collection<? extends E> c)`	Constructs a new linked hash set with the same elements as the specified collection.
`LinkedHashSet(int initialCapacity)`	Constructs a new, empty linked hash set with the specified initial capacity and the default load factor (0.75).
`LinkedHashSet(int initialCapacity, float loadFactor)`	Constructs a new, empty linked hash set with the specified initial capacity and load factor

- How LinkedHashSet implements LinkedList ? i.e. how it take care of order of insertion ? Because its syntax shows its extends HashSet and implements Set both of which don't provide such behaviour. Here is how it works.

  If we check the source for LinkedHashSet file, we find that it contains only implementation of above four constrcutors. But each implementation of the constructor is calling a super constructor.

  Example :

```
public LinkedHashSet(int paramInt, float paramFloat)
   {
         super(paramInt, paramFloat, true);
    }
```

  Which is the fourth constructor listed below. Here we can see that super constructor taking a Boolean value 'true' as one of the parameter. But SuperClass HashSet don't have such constructor. Actually HashSet.java file defines one constructor only for the purpose of LinkedHashSet which is shown below.

```
HashSet(int paramInt, float paramFloat, boolean paramBoolean)
   {
         this.map = new LinkedHashMap(paramInt, paramFloat);
   }
```

From above its clear that LinkedHashSet like HashSet internally used instance of LinkedHashMap as a has-a-relationship. That is the key.

25. TreeSet :

- Syntax : `public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Cloneable, Serializable`
- Elements of TreeSet are arranged in their natural order(due to comparable interface as explained below) or according to the comparator provided during set creation.
- Constructors

Constructor and Description
TreeSet() Constructs a new, empty tree set, sorted according to the natural ordering of its elements.
TreeSet(**Collection**<? extends **E**> c) Constructs a new tree set containing the elements in the specified collection, sorted according to the *natural ordering* of its elements.
TreeSet(**Comparator**<? super **E**> comparator) Constructs a new, empty tree set, sorted according to the specified comparator.
TreeSet(**SortedSet**<**E**> s) Constructs a new tree set containing the same elements and using the same ordering as the specified sorted set.

- TreeSet is not synchronized and its better to synchronize it during creation if more than one thread going to access it. The better way to synchronize is to use `Collections.synchronizedSortedSet`
- Example :

```
package collections;

import java.util.*;

public class TreeSetClass
{
```

```
public static void main(String[] args)
{
	TreeSet<Integer> tsc = new TreeSet<Integer>();

	//adding element to first treeset tsc
	tsc.add(5);
	tsc.add(7);
	tsc.add(3);
	tsc.add(2);
	tsc.add(6);
	tsc.add(3);

	System.out.println("Elements of tree set "+tsc);

	//methods related to SortedSet
	System.out.println("--------SortedSet Methods-------");

	//first and last
	System.out.println("first element in tsc is "+tsc.first());
	System.out.println("last element in tsc is "+tsc.last());

	//headSet, tailSet and subSet
	System.out.println("head set for 6 "+tsc.headSet(6));
	System.out.println("tail set for 5 "+tsc.tailSet(5));
	System.out.println("subset between 2 and 6 "+tsc.subSet(2,6));


	//methods related to NavigableSet
	System.out.println("--------NavigableSet Methods-------");

	//descendingIterator and decendingSet
	System.out.println("printing set using descending iterator");
	Iterator<Integer> it = tsc.descendingIterator();
	while(it.hasNext())
	{
		Integer element = it.next();
		System.out.println(element);
	}
	System.out.println("Descending Set is "+tsc.descendingSet());

	//floor and ceiling
	System.out.println("ceiling value of 4 "+tsc.ceiling(4));
	System.out.println("ceiling value of 8 "+tsc.ceiling(8));
	System.out.println("floor value of 2 "+tsc.floor(2));
	System.out.println("floor value of 1 "+tsc.floor(1));

	//headSet, tailSet and subset
	System.out.println("head set for 5 "+tsc.headSet(5, true));
	System.out.println("head set for 5 "+tsc.headSet(5, false));
	System.out.println("tail set for 3 "+tsc.tailSet(3, true));
	System.out.println("tail set for 3 "+tsc.tailSet(3, false));
	    System.out.println("subset for 3 and 7 is "+tsc.subSet(3,
	    true, 7, false));

	//higher and lower
	System.out.println("higher element than 6 is "+tsc.higher(6));
	System.out.println("higher element than 7 is "+tsc.higher(7));
	System.out.println("lower element than 4 is "+tsc.lower(4));
	System.out.println("lower element than 2 is "+tsc.lower(2));

	//pollFirst and pollLast
```

```
            tsc.pollFirst();
            System.out.println("after pollFirst "+tsc);
            tsc.pollLast();
            System.out.println("after pollLast "+tsc);
        }
    }
```

Output:

```
Elements of tree set [2, 3, 5, 6, 7]
--------SortedSet Methods-------
first element in tsc is 2
last element in tsc is 7
head set for 6 [2, 3, 5]
tail set for 5 [5, 6, 7]
subset between 2 and 6 [2, 3, 5]
--------NavigableSet Methods-------
printing set using descending iterator
7
6
5
3
2
Descending Set is [7, 6, 5, 3, 2]
ceiling value of 4 5
ceiling value of 8 null
floor value of 2 2
floor value of 1 null
head set for 5 [2, 3, 5]
head set for 5 [2, 3]
tail set for 3 [3, 5, 6, 7]
tail set for 3 [5, 6, 7]
subset for 3 and 7 is [3, 5, 6]
higher element than 6 is 7
higher element than 7 is null
lower element than 4 is 3
lower element than 2 is null
after pollFirst [3, 5, 6, 7]
after pollLast [3, 5, 6]
```

Comparable<T> and natural ordering :

- If there is a list 'l' then it can be sorted as below
  Collections.sort(l);
- If list consists of String elements then it will be sorted in alphabetical order, if its Date elements then it will be sorted in Chronological order etc which are called as natural ordering of elements.
- How Collections.Sort() works ?
  1. When it is called then it converts list to arrays by calling toArray() method on list.
  2. When array 'a' is obtained which contains all list elements then Arrays.sort(a) method is called.
  3. Arrays class contains sort methods for almost all kinds of primitives arrays.
  4. So whichever is the element type of list, example String that particular sort method is called.
  5. Now another requirement is each element type should implement Comparable<T> interface. This interface contains single method compareTo(T o) which defines the natural ordering of the element.

6. So now inside Arrays.sort() method particular compareTo() method is called to get their natural ordering.
7. Below are list of classes which implements Comparable<T> interface with their natural ordering.

Class	Natural Ordering
Byte	Signed numerical
Character	Unsigned numerical
Long	Signed numerical
Integer	Signed numerical
Short	Signed numerical
Double	Signed numerical
Float	Signed numerical
BigInteger	Signed numerical
BigDecimal	Signed numerical
Boolean	Boolean.FALSE < Boolean.TRUE
File	System-dependent lexicographic on path name
String	Lexicographic
Date	Chronological
CollationKey	Locale-specific lexicographic

- If one want to use ordering other than natural ordering then one can use another interface Comparator.

```
public interface Comparator<T>
   {
     int compare(T o1, T o2);
   }
```

- compare() method has two arguments returning negative, zero or positive integer same as compareTo().

Cloneable interface :

- This interface doesn't contain any method but interface itself is a indication of Object.clone() method. The interface Cloneable under java.lang is as

```
public interface Cloneable
 {

 }
```

i.e. in reality class which implements Cloneable actually override Object.clone() method. By convention, classes that implement this interface should override Object.clone (which is protected) with a public method.

26. Map Interfaces :

Interface	Description
Map	Maps unique keys to values.
Map.Entry	Describes an element (a key/value pair) in a map. This is an inner class of **Map**.
NavigableMap	Extends **SortedMap** to handle the retrieval of entries based on closest-match searches.
SortedMap	Extends **Map** so that the keys are maintained in ascending order.

- Map Interface

```
public interface Map<K,V>
```

Method	Description
void clear( )	Removes all key/value pairs from the invoking map.
boolean containsKey(Object *k*)	Returns **true** if the invoking map contains *k* as a key. Otherwise, returns **false**.
boolean containsValue(Object *v*)	Returns **true** if the map contains *v* as a value. Otherwise, returns **false**.
Set<Map.Entry<K, V>> entrySet( )	Returns a **Set** that contains the entries in the map. The set contains objects of type **Map.Entry**. Thus, this method provides a set-view of the invoking map.
boolean equals(Object *obj*)	Returns **true** if *obj* is a **Map** and contains the same entries. Otherwise, returns **false**.
V get(Object *k*)	Returns the value associated with the key *k*. Returns **null** if the key is not found.
int hashCode( )	Returns the hash code for the invoking map.

boolean isEmpty( )	Returns **true** if the invoking map is empty. Otherwise, returns **false**.
Set<K> keySet( )	Returns a **Set** that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.
V put(K *k*, V *v*)	Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are *k* and *v*, respectively. Returns **null** if the key did not already exist. Otherwise, the previous value linked to the key is returned.
void putAll(Map<? extends K, ? extends V> *m*)	Puts all the entries from *m* into this map.
V remove(Object *k*)	Removes the entry whose key equals *k*.
int size( )	Returns the number of key/value pairs in the map.
Collection<V> values( )	Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

- Map stores entries in key/value pairs where both keys and values are objects. Keys must be unique where values can be duplicated.
- Maps don't implement iterable interface hence its not possible to iterate through maps. However, after getting collection-view of maps its possible to use either iterator or use for-loop.
- Maps are part of collection framework although they don't implement collection interface. One can have collection-view of map using entrySet(), keyset() and values() methods as mentioned above.

27. SortedMap :

- `public interface SortedMap<K,V> extends Map<K,V>`
- SortedMap extends Map and ensures that all its entries are maintained in ascending order according to keys natural ordering, or according to a comparator provided at the time of SortedMap creation.
- Methods defined by SortedMap are summarized below

Method	Description
Comparator<? super K> comparator( )	Returns the invoking sorted map's comparator. If natural ordering is used for the invoking map, **null** is returned.
K firstKey( )	Returns the first key in the invoking map.
SortedMap<K, V> headMap(K *end*)	Returns a sorted map for those map entries with keys that are less than *end*.
K lastKey( )	Returns the last key in the invoking map.
SortedMap<K, V> subMap(K *start*, K *end*)	Returns a map containing those entries with keys that are greater than or equal to *start* and less than *end*.
SortedMap<K, V> tailMap(K *start*)	Returns a map containing those entries with keys that are greater than or equal to *start*.

28. NavigableMap :

- `public interface` **`NavigableMap<K,V>`** `extends SortedMap<K,V>`
- NavigableMap declares the behaviour of a map that supports the retrieval of entries based on the closest match to a given key or keys.
-

Method	Description
Map.Entry<K,V> ceilingEntry(K *obj*)	Searches the map for the smallest key *k* such that *k* >= *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K ceilingKey(K *obj*)	Searches the map for the smallest key *k* such that *k* >= *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
NavigableSet<K> descendingKeySet( )	Returns a **NavigableSet** that contains the keys in the invoking map in reverse order. Thus, it returns a reverse set-view of the keys. The resulting set is backed by the map.
NavigableMap<K,V> descendingMap( )	Returns a **NavigableMap** that is the reverse of the invoking map. The resulting map is backed by the invoking map.
Map.Entry<K,V> firstEntry( )	Returns the first entry in the map. This is the entry with the least key.
Map.Entry<K,V> floorEntry(K *obj*)	Searches the map for the largest key *k* such that *k* <= *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K floorKey(K *obj*)	Searches the map for the largest key *k* such that *k* <= *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
NavigableMap<K,V> headMap(K *upperBound*, boolean *incl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting map is backed by the invoking map.
Map.Entry<K,V> higherEntry(K *obj*)	Searches the set for the largest key *k* such that *k* > *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K higherKey(K *obj*)	Searches the set for the largest key *k* such that *k* > *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
Map.Entry<K,V> lastEntry( )	Returns the last entry in the map. This is the entry with the largest key.
Map.Entry<K,V> lowerEntry(K obj)	Searches the set for the largest key *k* such that *k* < *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K lowerKey(K *obj*)	Searches the set for the largest key *k* such that *k* < *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.

NavigableSet<K> navigableKeySet( )	Returns a **NavigableSet** that contains the keys in the invoking map. The resulting set is backed by the invoking map.
Map.Entry<K,V> pollFirstEntry( )	Returns the first entry, removing the entry in the process. Because the map is sorted, this is the entry with the least key value. **null** is returned if the map is empty.
Map.Entry<K,V> pollLastEntry( )	Returns the last entry, removing the entry in the process. Because the map is sorted, this is the entry with the greatest key value. **null** is returned if the map is empty.
NavigableMap<K,V> subMap(K *lowerBound*, boolean *lowIncl*, K *upperBound* boolean *highIncl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are greater than *lowerBound* and less than *upperBound*. If *lowIncl* is **true**, then an element equal to *lowerBound* is included. If *highIncl* is **true**, then an element equal to *highIncl* is included. The resulting map is backed by the invoking map.
NavigableMap<K,V> tailMap(K *lowerBound*, boolean *incl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are greater than *lowerBound*. If *incl* is **true**, then an element equal to *lowerBound* is included. The resulting map is backed by the invoking map.

29. Map.Entry<K, V> Interface :

- public static interface **Map.Entry<K,V>**
- This interface enables one to work with Map entries. Recall the entrySet() method declared by map returns collection view of the map, whose elements are of this class.
- The only way to obtain reference to a map entry is from the iterator of this collection-view. These Map.Entry objects are valid only for the duration of the iteration i.e. the behaviour of the map entry is undefined if the backing map has been modified after the entry was returned by the iterator, except through the setValue operation of the map entry.
- Methods defined by Map.Entry interface are as below.

Method	Description
boolean equals(Object *obj*)	Returns **true** if *obj* is a **Map.Entry** whose key and value are equal to that of the invoking object.
K getKey( )	Returns the key for this map entry.
V getValue( )	Returns the value for this map entry.
int hashCode( )	Returns the hash code for this map entry.
V setValue(V *v*)	Sets the value for this map entry to *v*. A **ClassCastException** is thrown if *v* is not the correct type for the map. An **IllegalArgumentException** is thrown if there is a problem with *v*. A **NullPointerException** is thrown if *v* is **null** and the map does not permit **null** keys. An **UnsupportedOperationException** is thrown if the map cannot be changed.

30. Map Classes :

Class	Description
AbstractMap	Implements most of the **Map** interface.
EnumMap	Extends **AbstractMap** for use with **enum** keys.
HashMap	Extends **AbstractMap** to use a hash table.
TreeMap	Extends **AbstractMap** to use a tree.
WeakHashMap	Extends **AbstractMap** to use a hash table with weak keys.
LinkedHashMap	Extends **HashMap** to allow insertion-order iterations.
IdentityHashMap	Extends **AbstractMap** and uses reference equality when comparing documents.

AbstractMap is a super class of all the concrete map class implementation.

- There are three general purpose map classes HashMap, LinkedHashMap and TreeMap. HashMap doesn't guarantee about the order of its elements, LinkedHashMap is nothing but a hash map with a linked list running through it. TreeMap like tree set order its elements in their natural order or according to the comparator provided.
- Example :

```
package collections;

import java.util.*;

public class DifferentFlavoursOfMaps {

   public static void main(String[] args)
   {
          HashMap<String, String> hm = new HashMap<String, String>();
          hm.put("jj", "nine");
          hm.put("jay", "five");
          hm.put("how", "seven");
          hm.put("fin", "one");
          hm.put("good", "three");

          System.out.println("Hash map contains "+hm);

          LinkedHashMap<Integer, String> lhm = new LinkedHashMap<Integer,
          String>();
          lhm.put(9, "nine");
          lhm.put(5, "five");
          lhm.put(7, "seven");
          lhm.put(1, "one");
          lhm.put(3, "three");
          System.out.println("Linked hash map contains "+lhm);

          TreeMap<String, String> tm1 = new TreeMap<String, String>();
          tm1.putAll(hm);
          System.out.println("Tree map contains "+tm1);

          TreeMap<Integer, String> tm2 = new TreeMap<Integer, String>();
          tm2.putAll(lhm);
```

```
            System.out.println("Tree map contains "+tm2);
    }
}
Output:
Hash map contains {how=seven, good=three, fin=one, jay=five, jj=nine}
Linked hash map contains {9=nine, 5=five, 7=seven, 1=one, 3=three}
Tree map contains {fin=one, good=three, how=seven, jay=five, jj=nine}
Tree map contains {1=one, 3=three, 5=five, 7=seven, 9=nine}
```

31. HashMap Class :

- `public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable`
- HashMap is HashTable based implementation of the map. HashMap class is roughly equivalent of HashTable except it is unsynchronized and permits null. i.e. HashMap permits both null values and null keys.
- HashMap makes no guarantee about the order of the map.
- Constructors :

Constructor and Description
HashMap()   Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).
HashMap(int initialCapacity)   Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75).
HashMap(int initialCapacity, float loadFactor)   Constructs an empty HashMap with the specified initial capacity and load factor.
HashMap(**Map**<? extends **K**,? extends **V**> m)   Constructs a new HashMap with the same mappings as the specified Map.

- Capacity is nothing but number of buckets in the Hash table and initial capacity is nothing but capacity of the hash map when it is created. The load factor is measure of how full the hash table is allowed to get before its capacity is automatically increased. By default load factor is 0.75. When number of entries in hash table exceeds the product of load factor and the current capacity then the hash table is rehashed. So that the hash table will have twice the number of buckets.

- Example :

```
package collections;

import java.util.*;

public class HashMapClass
{
   public static void main(String[] args)
   {
         HashMap<Integer,String> hmc = new HashMap<Integer,String>();
         hmc.put(5, "five");
         hmc.put(5, "five");
         hmc.put(1, "one");
         hmc.put(8, "eight");
         hmc.put(3, "three");
         hmc.put(6, "six");
         hmc.put(11, "elevan");
         hmc.put(2, "two");
         hmc.put(null, "Null");
         hmc.put(4, "four");

         System.out.println(hmc);

         //get method
         System.out.println("value for key 8 is "+hmc.get(8));

         //containsKey and containsValue
         System.out.println("contains key 11? "+hmc.containsKey(11));
         System.out.println("contains value Null?
         "+hmc.containsValue("Null"));
         System.out.println("contains value jj?
         "+hmc.containsValue("jj"));

         //size and isEmpty
         System.out.println("size of hash map - "+hmc.size());
         System.out.println("hash map is empty? "+hmc.isEmpty());

         //entrySet, keySet, values
         System.out.println("entry set with key-value "+hmc.entrySet());
         System.out.println("key set "+hmc.keySet());
         System.out.println("only values "+hmc.values());

         //remove and clear
       hmc.remove(4);
       System.out.println("hash map after removing key 4 "+hmc);
       hmc.clear();
       System.out.println("hash map after clear - "+hmc);
   }
}
Output:
{null=Null, 1=one, 2=two, 3=three, 4=four, 5=five, 6=six, 8=eight,
11=elevan}
value for key 8 is eight
contains key 11? true
contains value Null? true
contains value jj? false
size of hash map - 9
hash map is empty? false
entry set with key-value [null=Null, 1=one, 2=two, 3=three, 4=four,
5=five, 6=six, 8=eight, 11=elevan]
```

```
key set [null, 1, 2, 3, 4, 5, 6, 8, 11]
only values [Null, one, two, three, four, five, six, eight, elevan]
hash map after removing key 4 {null=Null, 1=one, 2=two, 3=three,
5=five, 6=six, 8=eight, 11=elevan}
hash map after clear - {}
```

32. TreeMap Example :

```
package collections;

import java.util.*;

public class TreeMapClass
{
   public static void main(String[] args)
   {
         TreeMap<Integer, String> tm = new TreeMap<Integer, String>();
         tm.put(4, "four");
         tm.put(2, "two");
         tm.put(10, "ten");
         tm.put(6, "six");
         tm.put(8, "eight");

         System.out.println("Content of tree map are "+tm);

         //SortedMap methods
         System.out.println("*****SortedMap Methods********");

         //firstKey and lastKey
         System.out.println("-----firstKey and lastKey-----");
         System.out.println("first key in tm is "+tm.firstKey());
         System.out.println("last key in tm is "+tm.lastKey());

         //headMap, tailMap and subMap
         System.out.println("-----headMap, tailMap and subMap-----");
         System.out.println("head map for 6 "+tm.headMap(6));
         System.out.println("tail map for 8 "+tm.tailMap(8));
         System.out.println("sub map for 3 and 8"+tm.subMap(3, 8));

         //NavigableMap methods
         System.out.println("*****NavigableMap Methods*******");

         //descending keyset and descending map
         System.out.println("-----descending keyset and map------");
         System.out.println("descending keySet "+tm.descendingKeySet());
         System.out.println("descending map "+tm.descendingMap());

         //ceiling and floor both keys and entries
         System.out.println("-----ceiling and floor keys and entries----
         -");
         System.out.println("Ceiling key and entry for 2 is
         "+tm.ceilingKey(2)+" and "+tm.ceilingEntry(2));
         System.out.println("Ceiling key and entry for 5 is
         "+tm.ceilingKey(5)+" and "+tm.ceilingEntry(5));
         System.out.println("Floor key and entry for 9 is
         "+tm.floorKey(9)+" and "+tm.floorEntry(9));
```

```
        //higher and lower both keys and entries
        System.out.println("-----higher and lower keys and entries-----
        ");
        System.out.println("highest keys and entries for 5 is
        "+tm.higherKey(5)+" and "+tm.higherEntry(5));
        System.out.println("lowest keys and entries for 9 is
        "+tm.lowerKey(9)+" and "+tm.lowerEntry(9));

        //headMap, tailMap and subMap
        System.out.println("-----headMap, tailMap and subMap-----");
        System.out.println("head map for 7 is "+tm.headMap(7, true));
        System.out.println("tail map for 4 is "+tm.tailMap(4, true));
        System.out.println("sub map for 3 and 10 is "+tm.subMap(3,
        false, 10, false));

        //first and last both keys and entries
        System.out.println("------first and last keys and entries------
        -");
        System.out.println("first key is "+tm.firstKey());
        System.out.println("first entry is "+tm.firstEntry());
        System.out.println("last key is "+tm.lastKey());
        System.out.println("last entry is "+tm.lastEntry());

        //Navigable set
        System.out.println("------Navigable set------");
        System.out.println("Navigable key set "+tm.navigableKeySet());

        //pollfirst and polllast methods
        System.out.println("------pollFirst and pollLast------");
        tm.pollFirstEntry();
        System.out.println("after pollFirst entry "+tm);
        tm.pollLastEntry();
        System.out.println("after pollLast entry "+tm);

    }
}
```

```
Output:
Content of tree map are {2=two, 4=four, 6=six, 8=eight, 10=ten}
*****SortedMap Methods********
-----firstKey and lastKey-----
first key in tm is 2
last key in tm is 10
-----headMap, tailMap and subMap-----
head map for 6 {2=two, 4=four}
tail map for 8 {8=eight, 10=ten}
sub map for 3 and 8{4=four, 6=six}
*****NavigableMap Methods*******
-----descending keyset and map------
descending keySet [10, 8, 6, 4, 2]
descending map {10=ten, 8=eight, 6=six, 4=four, 2=two}
-----ceiling and floor keys and entries-----
Ceiling key and entry for 2 is 2 and 2=two
Ceiling key and entry for 5 is 6 and 6=six
Floor key and entry for 9 is 8 and 8=eight
-----higher and lower keys and entries-----
highest keys and entries for 5 is 6 and 6=six
lowest keys and entries for 9 is 8 and 8=eight
-----headMap, tailMap and subMap-----
head map for 7 is {2=two, 4=four, 6=six}
tail map for 4 is {4=four, 6=six, 8=eight, 10=ten}
sub map for 3 and 10 is {4=four, 6=six, 8=eight}
```

```
------first and last keys and entries-------
first key is 2
first entry is 2=two
last key is 10
last entry is 10=ten
------Navigable set------
Navigable key set [2, 4, 6, 8, 10]
------pollFirst and pollLast------
after pollFirst entry {4=four, 6=six, 8=eight, 10=ten}
after pollLast entry {4=four, 6=six, 8=eight}
```

Chaptter 12. Input Output

1. Types of Input Streams Classes :

Input stream's job is to represent classes that produce input from different sources. These sources could be

1) An array of bytes.
2) A **String** object.
3) A file.
4) A "pipe," which works like a physical pipe: You put things in at one end and they come out the other.
5) A sequence of other streams, so you can collect them together into a single stream.
6) Other sources, such as an Internet connection.

Hence we can see because of different sources of input we have corresponding classes.

Class	Function	Constructor arguments How to use it
ByteArray-InputStream	Allows a buffer in memory to be used as an **InputStream.**	The buffer from which to extract the bytes. As a source of data: Connect it to a **FilterInputStream** object to provide a useful interface.
StringBuffer-InputStream	Converts a **String** into an **InputStream.**	A **String**. The underlying implementation actually uses a **StringBuffer**. As a source of data: Connect it to a **FilterInputStream** object to provide a useful interface.
File-InputStream	For reading information from a file.	A **String** representing the file name, or a **File** or **FileDescriptor** object. As a source of data: Connect it to a **FilterInputStream** object to provide a useful interface.
Piped-InputStream	Produces the data that's being written to the associated **PipedOutput-Stream**. Implements the "piping" concept.	**PipedOutputStream** As a source of data in multithreading: Connect it to a **FilterInputStream** object to provide a useful interface.
Sequence-InputStream	Converts two or more **InputStream** objects into a single **InputStream**.	Two **InputStream** objects or an **Enumeration** for a container of **InputStream** objects. As a source of data: Connect it to a **FilterInputStream** object to provide a useful interface.
Filter-InputStream	Abstract class that is an interface for decorators that provide useful functionality to the other **InputStream** classes. See Table I/O-3.	See Table I/O-3. See Table I/O-3.

1.1 Above Class **StringBufferInputStream** has been deprecated.

1.2 ByteArrayInputStream and ByteArrayOutputStream classes :

➔ This class implements an output stream in which the data is written into a byte array. The buffer automatically grows as data is written to it. The data can be retrieved using toByteArray()and toString().

Constructor and Description
ByteArrayInputStream(byte[] buf) Creates a ByteArrayInputStream so that it uses buf as its buffer array.
ByteArrayInputStream(byte[] buf, int offset, int length) Creates ByteArrayInputStream that uses buf as its buffer array.

Example 1 : Reading from console.

```
package inputOutputStream;

import java.io.*;

public class ByteArrayStreamCls {

   public static void main(String args[])throws IOException {

      ByteArrayOutputStream bOutput = new ByteArrayOutputStream(12);

      while( bOutput.size()!= 10 ) {
         // Gets the inputs from the user
         bOutput.write(System.in.read());
      }

      byte b [] = bOutput.toByteArray();
      System.out.println("Print the content");
      for(int x= 0 ; x < b.length; x++) {
         // printing the characters
         System.out.print((char)b[x]  + "   ");
      }
      System.out.println("   ");

      int c;

      ByteArrayInputStream bInput = new ByteArrayInputStream(b);

      System.out.println("Converting characters to Upper case " );
      for(int y = 0 ; y < 1; y++ ) {
         while(( c= bInput.read())!= -1) {
            System.out.println(Character.toUpperCase((char)c));
         }
         bInput.reset();
      }
   }
}
Output:
Hello Welcome
Print the content
```

```
H   e   l   l   o       W   e   l   c
Converting characters to Upper case
H
E
L
L
O

W
E
L
C
```

➔ ByteArrayInputStream only takes an 'Byte Array' as argument. Hence every input needs to be converted to a byte array before passing to this stream.

➔ In above example bOutput.write() method will writes the input data into byte array which could be retrived using toByteArray() and toString() methods. In above example how much bytes of data should go into buffer will be defined by bOutput.size().

➔ There is a write method in class ByteArrayOutputStream with arguments **write(byte[] b, int off, int len)**. Here we can directly write the byte array at offset **off** and length **len**. This method needs ready byte[] as argument and is useful when one wants to 'write' to a file or append a string.

➔ System.in.read() where 'in' is a InputStream which has read() method to read from console. This input will be stored in byte array by ByteArrayOutputStream.

➔ When we send argument as 'System.in' it means stream which has such argument will read from keyboard(console) directly and if argument is 'System.in.read()' it means InputStream defined in System class will read the input data.

➔ Read() method has <u>return type</u> as 'int' and write method has <u>argument type</u> as 'int'. Hence to read from console we have to use ByteArrayOutputStream class.

➔ While reading from String, one can convert it to a byte array and send it to ByteArrayInputStream(or CharArrayWriter) without sending it to ByteArrayOutputStream(or CharWriter). But while reading from console(or keyboard) one dont have any other option of writing first into the byte(or char) array and then read it.

Example 2 : Reading from a String

```
package inputOutputStream;

import java.io.*;

public class ByteArrayStringCls {

	public static void main(String[] args) throws IOException
	{
		String str = "Hello World";
		int c1;

		byte [] b = str.getBytes();

		ByteArrayInputStream bi = new ByteArrayInputStream(b);

			while ((c1 = bi.read()) != -1)
			{
				System.out.println((char)c1);
			}
		bi.close();
	}
}
Output:
H
e
l
l
o

W
o
r
l
d
```

1.3 FileInputStream and FileOutputStream classes :

➔ A FileInputStream obtains input bytes from a file in a file system. What files are available depends on the host environment. FileInputStream is meant for reading streams of raw bytes such as image data. For reading streams of characters, consider using FileReader.

```
package inputOutputStream;

import java.io.*;

public class FileIOStreamClass
{
	public static void main(String[] args) throws IOException
	{
		FileInputStream fin = null;
		FileOutputStream fout = null;

		try
		{

		fin = new FileInputStream("F:/Java material/java pgm
		files/javapgmfile1.txt");
```

```
            fout = new FileOutputStream("F:/Java material/java pgm
            files/javapgmfile2.txt");

            int c;

            while((c = fin.read())!= -1)
            {
                  fout.write(c);
            }
            }finally{
                  if (fin != null) {
                fin.close();
            }
            if (fout != null) {
                fout.close();
            }
      }
   }
}
```

1.4 SequenceInputStream Class

➔ There is only SequenceInputStream is available with read() methods but no SequenceOutputStream present. Hence no write() method either.

➔ Constructors :

Constructor and Description
SequenceInputStream(Enumeration<? extends InputStream> e) Initializes a newly created SequenceInputStream by remembering the argument, which must be an Enumeration that produces objects whose run-time type is InputStream.
SequenceInputStream(InputStream s1, InputStream s2) Initializes a newly created SequenceInputStream by remembering the two arguments, which will be read in order, first s1 and then s2, to provide the bytes to be read from this SequenceInputStream.

➔ A SequenceInputStream represents the logical concatenation of other input streams. It starts out with an ordered collection of input streams and reads from the first one until end of file is reached.

Example :

```
package testProg;

import java.io.*;

public class SequenceStreamCls
{

      public static void main(String[] args) throws IOException
      {
            int c;
            FileInputStream fin1 = new FileInputStream("F:/Java material/java pgm
            files/SequenceStream1.txt");
            FileInputStream fin2 = new FileInputStream("F:/Java material/java pgm
            files/SequenceStream2.txt");

            FileOutputStream fout = new FileOutputStream("F:/Java material/java pgm
```

```
                files/SequenceStream3.txt");

                SequenceInputStream sqi = new SequenceInputStream(fin1,fin2);

                while((c = sqi.read()) != -1)
                {
                      fout.write(c);
                }

                fout.close();
                sqi.close();
                fin1.close();
                fin2.close();
        }
}

Output:
SequenceStream1.txt = hi there!!
SequenceStream2.txt = how r u?
SequenceStream3.txt = hi there!!how r u?
```

2. Types of Output Stream Classes :

This category includes the classes that decide where your output will go: an array of bytes (but not a String—presumably, you can create one using the array of bytes), a file, or a "pipe."

Class	Function	Constructor arguments How to use it
ByteArray-OutputStream	Creates a buffer in memory. All the data that you send to the stream is placed in this buffer.	Optional initial size of the buffer.
		To designate the destination of your data: Connect it to a **FilterOutputStream** object to provide a useful interface.
File-OutputStream	For sending information to a file.	A **String** representing the file name, or a **File** or **FileDescriptor** object.

		To designate the destination of your data: Connect it to a **FilterOutputStream** object to provide a useful interface.
Piped-OutputStream	Any information you write to this automatically ends up as input for the associated **PipedInput-Stream.** Implements the "piping" concept.	**PipedInputStream**
		To designate the destination of your data for multithreading: Connect it to a **FilterOutputStream** object to provide a useful interface.
Filter-OutputStream	Abstract class that is an interface for decorators that provide useful functionality to the other **OutputStream** classes. See Table 1/O-4-	See Table I/O-4.
		See Table I/O-4.

3. 3. Types of FilterInputStream Classes :

Data-InputStream	Used in concert with **DataOutputStream,** so you can read primitives **(int, char, long,** etc.) from a stream in a portable fashion.	**InputStream**
		Contains a full interface to allow you to read primitive types.
Buffered-InputStream	Use this to prevent a physical read every time you want more data. You're saying, "Use a buffer."	**InputStream,** with optional buffer size.
		This doesn't provide an interface per se. It just adds buffering to the process. Attach an interface object.
LineNumber-InputStream	Keeps track of line numbers in the input stream; you can call **getLineNumber( )** and **setLineNumber (int)**.	**InputStream**
		This just adds line numbering, so you'll probably attach an interface object.
Pushback-InputStream	Has a one-byte pushback buffer so that you can push back the last character read.	**InputStream**
		Generally used in the

4. <u>Type of FilterOutputStream Classes</u> :

Data-OutputStream	Used in concert with **DataInputStream** so you can write primitives (**int**, **char**, **long**, etc.) to a stream in a portable fashion.	**OutputStream** Contains a full interface to allow you to write primitive types.
PrintStream	For producing formatted output. While **DataOutputStream** handles the *storage* of data, **PrintStream** handles *display*.	**OutputStream,** with optional **boolean** indicating that the buffer is flushed with every newline. Should be the "final" wrapping for your **OutputStream** object. You'll probably use this a lot.
Buffered-OutputStream	Use this to prevent a physical write every time you send a piece of data. You're saying, "Use a buffer." You can call **flush( )** to flush the buffer.	**OutputStream,** with optional buffer size. This doesn't provide an interface per se. It just adds buffering to the process. Attach an interface object.

➔ The PrintStream Class prints all primitive data types and String objects in view-able format. The two important methods in **PrintStream** are **print( )** and **println( )** which are overloaded to print all the various types. The difference between **print( )** and **println( )** is that the latter adds a newline when it's done.

➔ You'll need to buffer your input almost every time, regardless of the I/O device you're connecting to, so it would have made more sense for the I/O library to have a special case (or simply a method call) for unbuffered input rather than buffered input. (Same goes for Buffered Output).

1.5 <u>FilterInputStream and FilterOutputStream</u>

1.5.1 <u>DataInputStream and DataOutputStream classes</u>

➔ A data input stream lets an application read primitive Java data types(`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`) as well as String values, from an underlying input stream in a machine-independent way. <u>An application uses a data output stream to write data that could later be read by a data input stream.</u>

➔ Syntax : public class DataInputStream extends FilterInputStream implements DataInput

➔ Example 1

```
package inputOutputStream;

import java.io.*;

public class DataStreamString {

	public static void main(String[] args) throws IOException
	{
		String dataFile = "F:/Java material/java pgm
files/DataOutputStream.txt";
		DataOutputStream dout = new DataOutputStream(new
FileOutputStream(dataFile));
		dout.writeUTF("Java");

		DataInputStream din = new DataInputStream(new
FileInputStream(dataFile));
		String k = din.readUTF();
		System.out.println(k);

		dout.close();
		din.close();
	}
}
Output:
Java
```

➔ Example 2:

```
package inputOutputStream;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.IOException;
import java.io.EOFException;

public class DataStreams {
    static final String dataFile = "F:/Java material/java pgm
files/invoicedata.txt";

    static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
    static final int[] units = { 12, 8, 13, 29, 50 };
    static final String[] descs = { "Java T-shirt",
            "Java Mug",
            "Duke Juggling Dolls",
            "Java Pin",
            "Java Key Chain" };

    public static void main(String[] args) throws IOException {

        DataOutputStream out = null;

        try {
            out = new DataOutputStream(new
                    BufferedOutputStream(new FileOutputStream(dataFile)));
```

```
            for (int i = 0; i < prices.length; i ++) {
                out.writeDouble(prices[i]);
                out.writeInt(units[i]);
                out.writeUTF(descs[i]);
            }
        } finally {
            out.close();
        }

        DataInputStream in = null;
        double total = 0.0;
        try {
            in = new DataInputStream(new
                    BufferedInputStream(new FileInputStream(dataFile)));

            double price;
            int unit;
            String desc;

            try {
                while (true) {
                    price = in.readDouble();
                    unit = in.readInt();
                    desc = in.readUTF();
                    System.out.format("You ordered %d units of %s at $%.2f%n",
                            unit, desc, price);
                    total += unit * price;
                }
            } catch (EOFException e) { }
            System.out.format("For a TOTAL of: $%.2f%n", total);
        }
        finally {
            in.close();
        }
    }
}
Output:
You ordered 12 units of Java T-shirt at $19.99
You ordered 8 units of Java Mug at $9.99
You ordered 13 units of Duke Juggling Dolls at $15.99
You ordered 29 units of Java Pin at $3.99
You ordered 50 units of Java Key Chain at $4.99
For a TOTAL of: $892.88
```

1.5.2 BufferedInputStream and BufferedOutputStream classes

➔ During unbuffered stream call read or write request will be handled directly by underlying OS. This can make program less efficient since each such request triggers disk access, network activity or some other operation that is relatively expensive. To reduce overhead, we have buffered I/O streams. Buffered input streams read data from a memory area known as buffer, the native API will be called only when the buffer is empty. Similarly, buffered output streams write data to a buffer and native output API is called only when buffer is full. There are four buffered stream classes used to wrap unbuffered streams: **BufferedInputStream** and **BufferedOutputStream** create buffered byte streams, while **BufferedReader** and **BufferedWriter** create buffered character streams. Above class DataStream.java shows how one should use BufferedInputStream and BufferedOutputStream.

1.5.3. LineNumberInputStream

This class has been deprecated. LineNumberReader Class is present. Please check the notes and examples.

The Predefined Streams :

- As we know, all java programs automatically import java.lang package. This package defines a class called **System**, which encapsulates several aspects of the run-time environment. It contains three predefined stream variables **in**, **out** and **err**. These fields are declared as public, static and final.

- **System.out** refers to the standard output stream. By default, this is the console. **System.in** refers to the standard input stream, which is the keyboard by default. **System.err** refers to the System error stream, which is also console by default.

- System.in is of object type **InputStream**; System.out and System.err are of object type **PrintStream**. These are byte streams, even though typically they are used to read and write characters from and to the console. Also they can be wrapped in character based stream, if desired. Three variables in, out, err defined in class 'System' as below

```
public final static InputStream in = null

public final static PrintStream out = null;

public final static PrintStream err = null;
```

- How System.out.println() works?

    - As we can see above that all stream variables are set to null inside System.java class. Then System.out.println() should throw NullPointerException. But in reality null value will be set to a new value. But how come a 'final' value can be changed ? It can be done through 'native' methods i.e. `setIn0(InputStream in), setOut0(PrintStream out), setErr0(PrintStream err).`

    - The above native methods are defined as

```
private static native void setIn0(InputStream in);
private static native void setOut0(PrintStream out);
private static native void setErr0(PrintStream err);
```

      Note that native methods are defined same as methods of interfaces but with 'native' keyword.

    - These methods are called from another method called initializeSystemClass() which is called by JVM after 'static' and 'thread' initialization. It is defined in System.java class as

```
private static void initializeSystemClass()

{

//

}
```

1.5.5 PrintStream

➔ A PrintStream adds functionality to another output stream, namely the ability to print representations of various data values conveniently. Two other features are provided as well. Unlike other output streams, a PrintStream never throws an IOException; instead, exceptional situations merely set an internal flag that can be tested via the checkError method. Optionally, a PrintStream can be created so as to flush automatically; this means that the flush method is automatically invoked after a byte array is written, one of the println methods is invoked, or a newline character or byte ('\n') is written.

➔ All characters printed by a PrintStream are converted into bytes using the platform's default character encoding. The PrintWriter class should be used in situations that require writing characters rather than bytes.

➔ Example

```
package inputOutputStream;

import java.io.*;

public class PrintStreamCls
{
	public static void main(String[] args) throws IOException
	{
		String str = "New String";
		String fileStream = "F:/Java material/java pgm files/PrintStream.txt";
		String fileWriter = "F:/Java material/java pgm files/PrintWriter.txt";

		//Printing Without flush method
		System.out.println("Prints nothing without flush method");
		PrintStream ps1 = new PrintStream(System.out);
		ps1.println(str);
		System.out.println("------");

		//Printing With flush method
		System.out.println("Printing with flush method");
		PrintStream ps2 = new PrintStream(System.out);
		ps2.println(str);
		ps2.flush();
		System.out.println("------");

		//Auto flush
		System.out.println("Auto flush");
		PrintStream ps3 = new PrintStream(System.out, true);
		ps3.println(str);
		System.out.println("------");
```

```
            //Append
            System.out.println("Appending string");
            PrintStream ps4 = new PrintStream(System.out);
            ps4.println(str);
            ps4.append('A');
            ps4.append("Jayant");
            ps4.flush();
            System.out.println();
            System.out.println("------");

            //Writing to a file using PrintStream
            //Writes bytes not characters
            PrintStream ps5 = new PrintStream(fileStream);
            ps5.write(12);
            ps5.flush();
            ps5.close();

            //Writing to a file using PrintWriter
            //Writers characters
            PrintWriter pw = new PrintWriter(fileWriter);
            pw.write("Hello World..");
            pw.flush();
            pw.close();
      }
}
Output:
Prints nothing without flush method
New String
------
Printing with flush method
New String
------
Auto flush
New String
------
Appending string
New String
AJayant
------
```

InputStreamReader and OutputStreamWriter :

➔ An **InputStreamReader** is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset.

➔ An **OutputStreamWriter** is a bridge from character streams to byte streams: Characters written to it are encoded into bytes using a specified charset.

➔ Each invocation of one of an InputStreamReader's read() methods may cause one or more bytes to be read from the underlying byte-input stream. To enable the efficient conversion of bytes to characters, more bytes may be read ahead from the underlying stream than are necessary to satisfy the current read operation. For top efficiency, consider wrapping an InputStreamReader within a BufferedReader. For example:

```
BufferedReader in  = new BufferedReader(new InputStreamReader(System.in));
```

➔ Each invocation of a write() method causes the encoding converter to be invoked on the given character(s). The resulting bytes are accumulated in a buffer before being written to the underlying output stream. The size of this buffer may be specified, but by default it is large enough for most purposes. Note that the characters passed to the write() methods are not buffered. For top efficiency, consider wrapping an OutputStreamWriter within a BufferedWriter so as to avoid frequent converter invocations. For example:

```
Writer out = new BufferedWriter(new OutputStreamWriter(System.out));
```

➔ Example

```
package testProg;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;


public class InputStreamReaderExample
{

    private static final String OUTPUT_FILE = "F:/Java material/java pgm
files/InputStreamReaderEx.txt";
    public static void main(String[] args)
    {
        char[] chars = new char[100];
        try
        {
            InputStreamReader inputStreamReader = new InputStreamReader(new
FileInputStream(OUTPUT_FILE),"UTF-8");
            // read 100 characters from the file
            while (inputStreamReader.read(chars) != -1)
                System.out.println(new String(chars));
        } catch (IOException e)
        {
          e.printStackTrace();
        }
     }
}
Output:
InputStreamReaderEx.txt: Hello world !! Today is a great day.
Console: Hello world !! Today is a great day.
```

5. Readers and Writers Classes :

➔ These classes has been added during Java 1.1 for Internationalization and not to replace InputStream and OutputStream Classes. Because the old I/O stream hierarchy supports only 8-bit byte streams and doesn't handle the 16-bit unicode characters well. Both **Reader** and **Writer** hierarchies supports unicodes in all I/O operations hence they are also called as Character Streams.

➔ There are times when you must use classes from the "byte" hierarchy *in combination* with classes in the "character" hierarchy. To accomplish this, there are "adapter" classes: **InputStreamReader** converts an **InputStream** to a **Reader**, and **OutputStreamWriter** converts an **OutputStream** to a **Writer**.

➔ Reader and Writer classes have all the corresponding I/O stream classes with the support for unicode.

➔ The package java.util.zip are byte oriented rather than char-oriented. So its better to use char-oriented I/O classes but there could be cases where one needs to use byte oriented classes(mostly during compiler gives error with char-oriented classes).

Sources & sinks: Java 1.0 class	Corresponding Java 1.1 class
InputStream	Reader adapter: InputStreamReader
OutputStream	Writer adapter: OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream (deprecated)	StringReader
(no corresponding class)	StringWriter
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter

➔ Reader and Writer Classes for Filter Stream Classes

Filters: Java 1.0 class	Corresponding Java 1.1 class
FilterInputStream	FilterReader
FilterOutputStream	FilterWriter (abstract class with no subclasses)
BufferedInputStream	BufferedReader (also has readLine( ))
BufferedOutputStream	BufferedWriter
DataInputStream	Use DataInputStream (except when you need to use readLine( ), when you should use a
	BufferedReader)
PrintStream	PrintWriter
LineNumberInputStream (deprecated)	LineNumberReader
StreamTokenizer	StreamTokenizer (Use the constructor that takes a Reader instead)
PushbackInputStream	PushbackReader

➔ Some Classes kept unchanged between java 1.0 and java 1.1 they are

Java 1.0 classes without corresponding Java 1.1 classes
DataOutputStream
File
RandomAccessFile
SequenceInputStream

5.1 CharArrayReader and CharArrayWriter Class :

➔ Example 1 : Reading from console

```
package inputOutputStream;
import java.io.*;

public class CharArrayClass {

	public static void main(String[] args)throws IOException
	{
		int i;
		CharArrayWriter cw = new CharArrayWriter();

		while( cw.size()!= 10 )
		{
			cw.write(System.in.read());
		}

		char [] c = cw.toCharArray();

		CharArrayReader cr = new CharArrayReader(c);

		while((i = cr.read())!= -1)
		{
			System.out.println((char)i);
		}
		cr.close();
	}
}
Output:
Hi how are you ?(input)
H
i

h
o
w

a
r
e
```

➔ Example 2 : Reading from a String

```
package inputOutputStream;
import java.io.*;

public class CharArrayString
{
	public static void main(String[] args) throws IOException
	{
		int i;
		String str = "Hello World";

		char [] c = str.toCharArray();

		CharArrayReader cr = new CharArrayReader(c);

		while((i = cr.read()) != -1)
		{
			System.out.println((char)i);
		}
	}
}
Output:
H
e
l
l
o

W
o
r
l
d
```

➔ Example 3 : Other methods of class CharArrayWriter like append() and writeto() has been used in the below example.

```
package inputOutputStream;

import java.io.*;

public class CharArrayOtherMeth {

	public static void main(String[] args) throws IOException
	{
		int i;
		char [] c1, c2;
		String str1 = "Hello how r u ?";
		String str2 = "I am fine";
		CharArrayOtherMeth cam = new CharArrayOtherMeth();
		CharArrayReader cr1, cr2;

		CharArrayWriter cw = new CharArrayWriter();

		//Reading from the writer
		cw.write(str1, 0, str1.length());
		c1 = cw.toCharArray();
		cr1 = new CharArrayReader(c1);

		System.out.println("Before append");
		cam.readMethod(cr1);
```

```
            //Appending the Writer
            cw.append(str2);
            c2 = cw.toCharArray();
            cr2 = new CharArrayReader(c2);

            System.out.println("After append");
            cam.readMethod(cr2);

            //Writing content of buffer to a file
            FileWriter fw = new FileWriter("F:/Java material/java pgm
            files/javapgmfile1.txt");
            cw.writeTo(fw);

            fw.close();
            cw.close();
      }

      void readMethod(CharArrayReader car) throws IOException
      {
            int c=0;
            while((c = car.read())!= -1)
            {
                  System.out.println((char)c);
            }
      }
}
Output:
Before append
H
e
l
l
o

h
o
w

r

u

?
After append
H
e
l
l
o

h
o
w

r

u

?
I

a
```

m

f
i
n
e

➔ Example 4

```
package inputOutputStream;

import java.io.*;

public class CharArrayStringPrint {

	public static void main(String[] args) throws IOException
	{
		String str;
		int c1, c2;
		char [] ch1;
		char [] ch2 = new char[1024];

		CharArrayWriter cw = new CharArrayWriter();

		while(cw.size()!= 10)
		{
			cw.write(System.in.read());
		}

		ch1 = cw.toCharArray();

		CharArrayReader cr1 = new CharArrayReader(ch1);
		CharArrayReader cr2 = new CharArrayReader(ch1);

		System.out.println("without string");
		while((c1 = cr1.read()) != -1)
		{
			System.out.println((char)c1);
		}

		System.out.println("with string");
		while((c2 = cr2.read(ch2)) != -1)
		{
			str = new String(ch2, 0, c2);
			System.out.println(str);
		}
	}
}
```

Output:

```
Hello World(console)
without string
H
e
l
l
o

W
o
r
```

```
l
with string
Hello Worl
```

How it works ?
We have the code

```
while((c2 = cr2.read(ch2)) != -1)
            {
                  str = new String(ch2, 0, c2);
                  System.out.println(str);
            }
```

Whats happening here is that cr2.read(ch2) will store input stream data into a char array 'ch2' and read(ch2) will return the number of characters read i.e. c2. Next a String 'str' will be constructed by using using ch2 and gets printed.

5.2 FileReader and FileWriter Class

```
package inputOutputStream;

import java.io.*;

public class FileReaderWriterClass {

      public static void main(String[] args) throws IOException
      {
            FileReader fr = null;
            FileWriter fw = null;

            try
            {
            fr = new FileReader("F:/Java material/java pgm files/CharReader1.txt");
            fw = new FileWriter("F:/Java material/java pgm files/CharReader2.txt");
                  int c;
            while((c = fr.read())!= -1)
            {
                  fw.write(c);
            }
            }
            finally
            {
                  if(fr != null)
                  {
                        fr.close();
                  }
                  if(fw != null)
                  {
                        fw.close();
                  }
            }
      }
}
```

6. FilterReader and FilterWriter Classes

6.1 BufferedReader and BufferedWriter Classes

Both works similar to BufferedInputStream and BufferedOutputStream classes. Please refer the notes and

check for examples. Only this classes extends FilterReader and FilterWriter classes respectively.

6.2 LineNumberReader Class

➔ A buffered Character input stream that keeps track of line numbers. This class defines methods setLineNumber(int) and getLineNumber() for setting and getting the current line number respectively.

➔ By default, line numbering begins at 0. This number increments at every line terminator as the data is read. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

➔ `public class LineNumberReader extends BufferedReader`

➔ Example :

```
package inputOutputStream;

import java.io.*;

public class LineNumberReaderCls
{
	public static void main(String [] args) throws IOException
	{
		int i;
		String str;
		String dataList = "F:/Java material/java pgm
		files/LineNumberReader.txt";

		try {
			FileReader fr = new FileReader(dataList);
			LineNumberReader lnr = new LineNumberReader(fr);

			//Read line till the end of the stream

			while((str=lnr.readLine()) != null)
			{
				i = lnr.getLineNumber();
				System.out.print("("+i+")");
				System.out.println(str);
			}
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
file:LineNumberReader.txt(contains)
apple
samsung
lg
nokia
htc

Output:
(1)apple
(2)samsung
(3)lg
(4)nokia
(5)htc
```

➔ setLineNumber() method

If we have one line

```
lnr.setLineNumber(20);
```

before while then output will be

```
Output:
(21)apple
(22)samsung
(23)lg
(24)nokia
(25)htc
```

As we can see, it will set line number to 20 and line reading will start from next line number i.e. 21. Hence we get the above output.

6.3 PushbackReader Class

➔ A character-stream reader that allows characters to be pushed back into the stream. When we read through a stream the control always goes to next byte/char. But with push back its possible to bring control back to the previous byte/char.

Constructor and Description
PushbackReader(Reader in) Creates a new pushback reader with a one-character pushback buffer.
PushbackReader(Reader in, int size) Creates a new pushback reader with a pushback buffer of the given size.

➔ Above we can see in the second constructor we can even define size for a push back buffer i.e. how many byte/char could be unread.

➔ Example

```
package inputOutputStream;

import java.io.*;

public class PushBackReaderCls {

	public static void main(String[] args) throws IOException
		{
		  char charArray[] = str.toCharArray();
		  CharArrayReader chARed = new CharArrayReader(charArray);

		  PushbackReader pbr = new PushbackReader(chARed);
		  int c;
		  try {
		      while( (c = pbr.read())!= -1) {
		          if(c == '-') {
		                int nextC;
```

```
                        if( (nextC = pbr.read()) == '-'){
                             System.out.print("**");
                        }else {
                              //push backs a single character
                            pbr.unread(nextC);
                             System.out.print((char)c);
                        }
                    }else {
                             System.out.print((char)c);
                    }
                }
            } catch(IOException ioe) {
                    System.out.println("Exception while reading" + ioe);
            }
        }
    }
    Output:
    a**b-cb**-ab
```

Above we can see that when two continuous hyphen(-) have not found then previous character will be pushed back and re-read.

➔ PushbackReader class have more unread methods as shown below

unread(char[] cbuf)

unread(char[] cbuf, int off, int len)

Through these methods one can unread array of characters or the portion of it(second method above). But if constructor has set the buffer limit than only that much will be allowed.

6.4 PrintWriter Class is same as PrintStream class. Please refer notes and examples.

Chapter 14 Serialization and File I/O

1. Introduction :

- Object have both **behaviour** and **state**. Behaviour lives within the class but state lives within each individual object.
- One does need to save the state of a game to play it from where he stopped or stop an video to start it from the same point. For all these we need to save the state of an object.
- Java provides many ways to save the state of an object.
    1. Serialization : Write the file that holds serialized objects. The programme can later read the objects from the file and inflate them back into living, breathing, heap inhabiting objects.
    2. Write a plain text file.
    3. Above two are not the only way off course. One can chose to write bytes instead of characters or any other primitive type etc.
- But regardless of method one use to save the state the fundamental I/O will be pretty much the same. Either one writes data to file disk or a stream coming from a network connection or Reading data through the same process in reverse i.e. reading either from file disk or a network conection.

2. Writing a serialized object to a file :

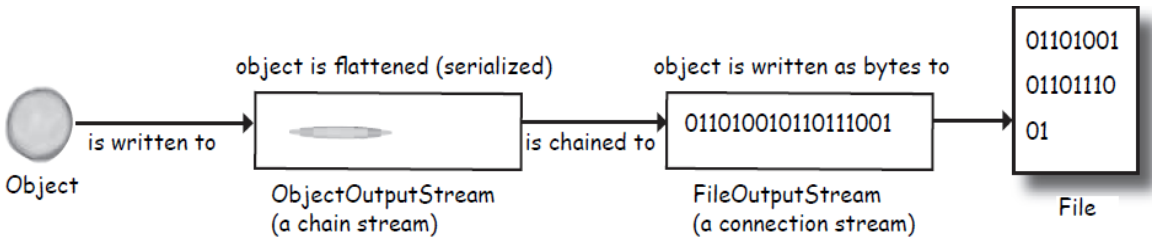
- Steps for serializing an object
    1. Make a FileOutputStream

       FileOutputStream fileStream = new FileOutputStream("MyGame.ser");
    2. Make an ObjectOutputStream

       ObjectOutputStream os = new ObjectOutputStream(fileStream);
    3. Write the Object

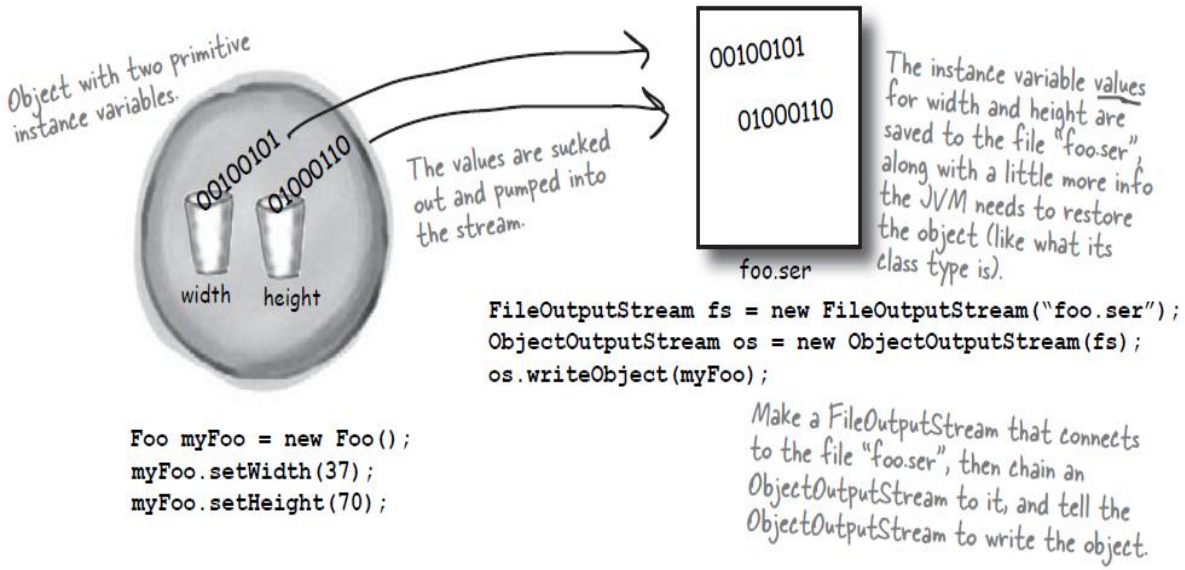
       os.writeObject(characterOne);
       os.writeObject(characterTwo);
       os.writeObject(characterThree);
    4. Close the ObjectOutputStream

       os.close();
- Always it will take two streams to hooked together to do something useful, this is called **Chaining**. Here we have ObjectOutputStream which turn object into data that can be written to a stream and FileOutputStream to write data to a file. Below diagram explains it correctly.

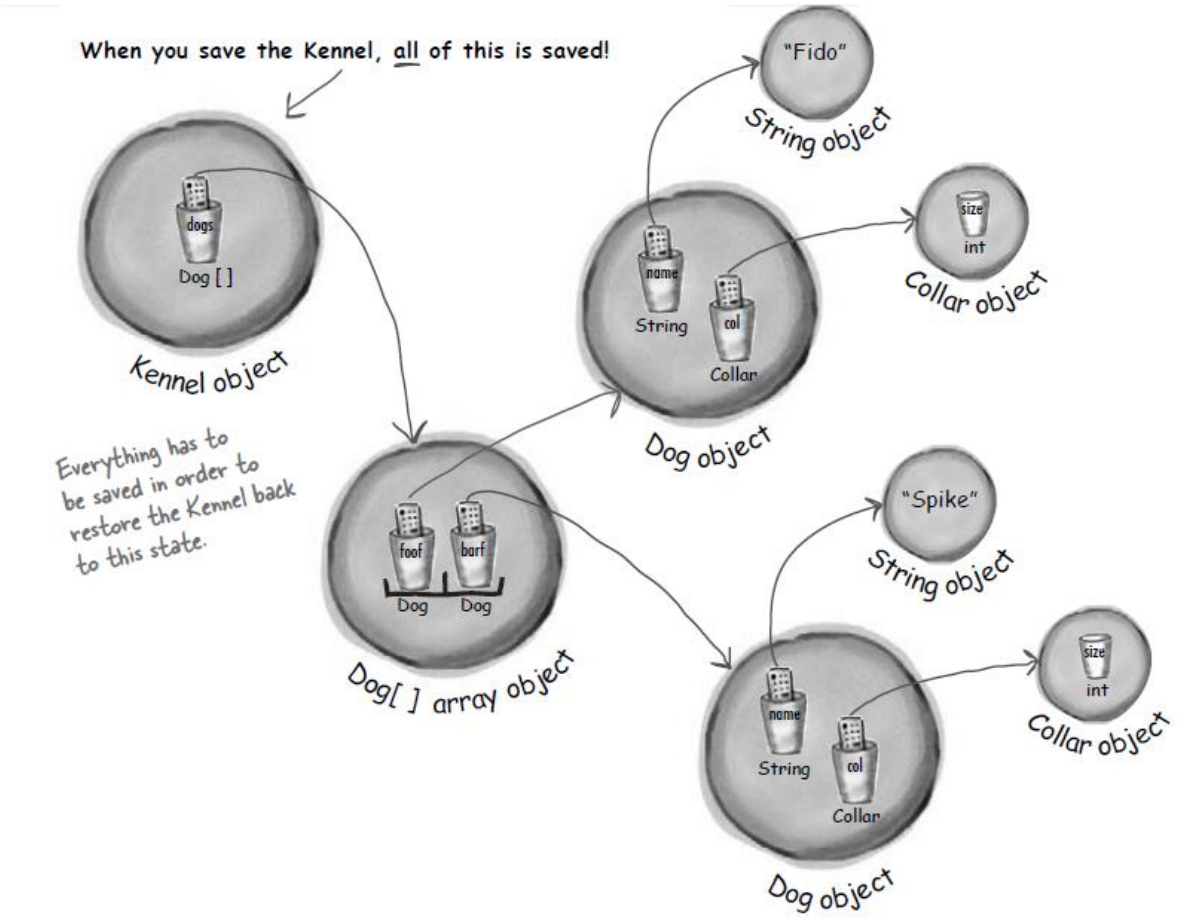


3. **<u>What happens when object is serialized</u>** :

- Object at heap have a state – the values of instance variables of object. These values make one instance of a class separate from the another instance of the same class.
- Serialized object saves the values of the instance variables so that an identical instance (object) can be brought back to life on the heap.
- Below diagram clearly explains it



- What exactly is an object's state ? that needs to be saved. Object can have a instance of a reference variable. That reference variable may inturn can have more reference variables. So during serialization all these objects i.e. object graph is serialized. Example if Kennel class have reference to Dog [] array object. The Dog[] array contains two Dog objects i.e. references of two Dog objects. Each Dog object hold references to a String and a Collar object. String is collection of characters and Collar contains int. When one tries to save Kennel i.e. Serialize Kennel then all of these will be saved. Below diagram explains it clearly.



4. **How to make a class Serializable** :
   - A class can be serializable if it implements **Serializable** interface. Serializable interface don't have any method, it is just a marker or tag interface which tells that the objects of that type are saveable through the serialization mechanism.
   - If any superclass of a class is Serializable then subclass is automatically serializable even if subclass doesn't explicitly declare '*implements Serializable'*.

5. **Class is not serialized whose reference is used**
   - Sometime it may happen that a class1 contains a reference variable for another class2(completely another class) and class2 don't implement serializable then during serialization compiler finds that class2 is not serializable hence throws 'NotSerializableException'. Example is shown below.

```
package inputOutputStream;

import java.io.*;
```

```
public class Pond implements Serializable
{
  private Duck duck = new Duck();
  public static void main (String[] args)
  {
        Pond myPond = new Pond();
        try
        {
              FileOutputStream fs = new FileOutputStream("Pond.ser");
              ObjectOutputStream os = new ObjectOutputStream(fs);
              os.writeObject(myPond);
              os.close();
        } catch(Exception ex)
              {
                    ex.printStackTrace();
              }
  }
}

class Duck
{
// duck code here
}
```

```
Output:
java.io.NotSerializableException: inputOutputStream.Duck
at inputOutputStream.Pond.main(Pond.java:14)
```

- To avoid such exception one should mark such field as '**transient**' keyword and the serialization will skip right over it. If we use transient keyword before duck reference as shown below then it will be skipped(from saving) during serialization and there will be no exception and programme executes correctly.

  ```
  private transient Duck duck = new Duck();
  ```

- Although most things in java class libraries are serializable still one cannot save many things like network connections, threads or file objects etc because they are dependent on a particular runtime experience and once the program shuts down there is no way to bring them back to life. They need to be created from scratch each time.
- If serializable is so useful then why by default every class implements serializable like if object class is made to implement serializable then it can be done. But it is no useful in many instances like one don't want to save a password object etc.
- If a class is not serializable i.e. don't implements Serializable then we can create a subclass of it and make it implements Serializable. This could be useful and then we can replace superclass instance with subclass. But when deserialization takes place then superclass constructor will run as new object is being created. (More on deserialization later).

- Similarly what happened to the variables which have been made transient during deserialization? The variable will be brought back as *null* regardless of the value it has during serialization. So what is the solution for such instances? There are two, first if the value is generic like dog has a collar but all collar objects are same then they can be reinitialize to the default value. Second is if value is contextual like this dog should have particular colour and design of collar then one should save the key values of the collar use them to recreate a brand new collar which is identical to original.
- What if two reference variables are holding the same object? Then it will be recognized in an Object graph that two objects are same and only one will be saved. During deserialization all references to that object are stored.

6. <u>Deserialization</u> :

- Deserialization is a lot like serialization in reverse.
- It will be done in following steps.

1. Make a FileInputStream

```
FileInputStream fileStream = new FileInputStream("MyGame.ser");
```

If file is not found then it will throw fileNotfoundException.

2. Make an ObjectInputStream

```
ObjectInputStream os = new ObjectInputStream(fileStream);
```

3. Read the objects

```
Object one = os.readObject();
Object two = os.readObject();
Object three = os.readObject();
```

Every time it calls readObject() then it will get the next object in the stream. So it will be read back in the same order they are written. Also if one tries to read more objects than what has been written then it will throw exception.

4. Cast the Objects

```
GameCharacter elf = (GameCharacter) one;
GameCharacter troll = (GameCharacter) two;
GameCharacter magician = (GameCharacter) three;
```

The return type of readObject() will be 'Object' hence they needed to be cast back to their original type.

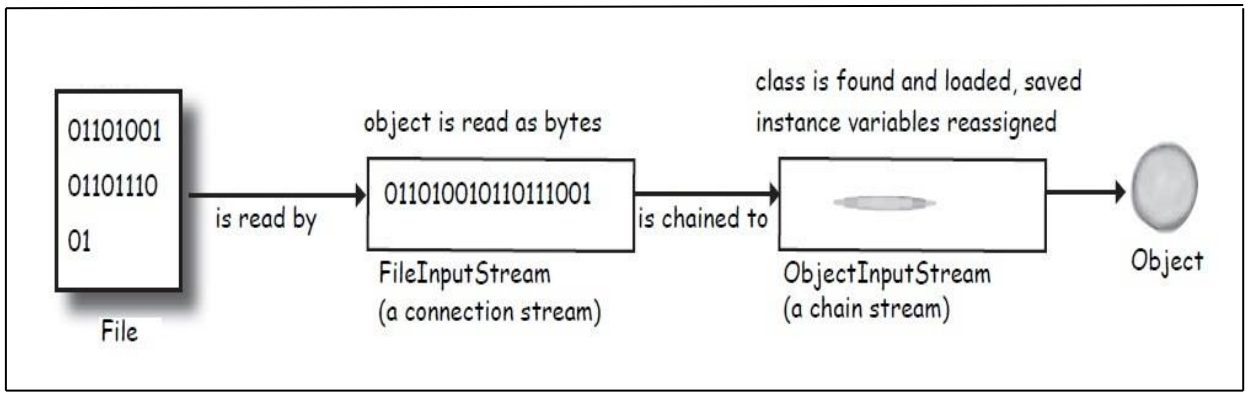
5. Close the ObjectInputStream

```
os.close();
```

Closing the stream at top closes the ones underneath i.e. FileInputStream(and file) will be closed automatically.

- **<u>What happens during deserialization</u>:**

When an Object is deserialized, JVM tries to bring back that object to life making a new object at the heap having the same state it had at the time of serialization. Except the transient variables which will be initialized as null (for object references) or default values(for primitives).



1. The object is **read** from the stream.

2. The JVM determines (through info stored with the serialized object) the object's **class type.**

3. The JVM attempts to **find and load** the object's **class**. If the JVM can't find and/or load the class, the JVM throws an exception and the deserialization fails.

4. A new object is given space on the heap, but the **serialized object's constructor does NOT run**! Obviously, if the constructor ran, it would restore the state of the object back to its original 'new' state, and that's not what we want. We want the object to be restored to the state it had *when it was serialized*, not when it was first created.

5. If the object has a non-serializable class somewhere up its inheritance tree, the **constructor for that non-serializable class will run** along with any constructors above that (even if they're serializable). Once the constructor chaining begins, you can't stop it, which means all superclasses, beginning with the first non-serializable one, will reinitialize their state.

6. The object's **instance variables are given the values from the serialized state**. Transient variables are given a value of null for object references and defaults (0, false, etc.) for primitives.

- Static variables are not encouraged for serialization as they dont belong to object but classes.

6. Example for both Serialization and Deserialization :

- Class GameCharacter :

```
package ch14SerializationAndFileIO;

import java.io.Serializable;

public class GameCharacter implements Serializable
{
    int power;
    String type;
    String[] weapons;

    public GameCharacter(int p, String t, String[] w)
    {
        power = p;
        type = t;
        weapons = w;
    }

    public int getPower() {
        return power;
    }

    public String getType() {
        return type;
    }

    public String getWeapons() {
        String weaponList = "";
        for (int i = 0; i < weapons.length; i++)
        {
            weaponList += weapons[i] + " ";
        }
        return weaponList;
    }
}
```

- Class GameSaverTest :

```
package ch14SerializationAndFileIO;

import java.io.*;

public class GameSaverTest
{
    public static void main (String[] args) {
        GameCharacter one = new GameCharacter(50, "Elf", new String[] {"bow",
"sword", "dust"});
```

```
        GameCharacter two = new GameCharacter(200, "Troll", new String[]
{"bare hands", "big axe"});
        GameCharacter three = new GameCharacter(120, "Magician", new String[]
{"spells", "invisibility"});

        try {
            ObjectOutputStream os = new ObjectOutputStream(new
FileOutputStream("D:/Game.txt"));
            os.writeObject(one);
            os.writeObject(two);
            os.writeObject(three);
            os.close();
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }

        one = null;
        two = null;
        three = null;

        try {
            ObjectInputStream is = new ObjectInputStream(new
FileInputStream("D:/Game.txt"));
            GameCharacter oneRestore = (GameCharacter) is.readObject();
            GameCharacter twoRestore = (GameCharacter) is.readObject();
            GameCharacter threeRestore = (GameCharacter) is.readObject();

            System.out.println("One's type: " + oneRestore.getType());
            System.out.println("Two's type: " + twoRestore.getType());
            System.out.println("Three's type: " + threeRestore.getType());
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
Output:
One's type: Elf
Two's type: Troll
Three's type: Magician
```

Note : If we mark 'type' as transient i.e.

```
transient String type;
```

in class GameCharacter.java then output would be

```
One's type: null
Two's type: null
Three's type: null
```

BULLET POINTS

- You can save an object's state by serializing the object.
- To serialize an object, you need an ObjectOutputStream (from the java.io package)
- Streams are either connection streams or chain streams
- Connection streams can represent a connection to a source or destination, typically a file, network socket connection, or the console.
- Chain streams cannot connect to a source or destination and must be chained to a connection (or other) stream.
- To serialize an object to a file, make a FileOuputStream and chain it into an ObjectOutputStream.
- To serialize an object, call *writeObject(theObject)* on the ObjectOutputStream. You do not need to call methods on the FileOutputStream.
- To be serialized, an object must implement the Serializable interface. If a superclass of the class implements Serializable, the subclass will automatically be serializable even if it does not specifically declare *implements Serializable*.
- When an object is serialized, its entire object graph is serialized. That means any objects referenced by the serialized object's instance variables are serialized, and any objects referenced by those objects...and so on.
- If any object in the graph is not serializable, an exception will be thrown at runtime, unless the instance variable referring to the object is skipped.
- Mark an instance variable with the *transient* keyword if you want serialization to skip that variable. The variable will be restored as null (for object references) or default values (for primitives).
- During deserialization, the class of all objects in the graph must be available to the JVM.
- You read objects in (using readObject()) in the order in which they were originally written.
- The return type of readObject() is type Object, so deserialized objects must be cast to their real type.
- Static variables are not serialized! It doesn't make sense to save a static variable value as part of a specific object's state, since all objects of that type share only a single value—the one in the class.

Chapter 14. Multithreaded Programming

1. Introduction

   - There are two types of multitasking one is process based and another is thread based. Process based multitasking is when computer runs two or three programmes concurrently ex. running a java compiler at the same time using a text editor or visiting a website. Thread based multitasking is when a single program can perform two or more task simultaneously. Ex. A text editor formats a text along with printing a page. Both tasks will be performed by two separate threads.

   - Process multitasking is heavy weight and thread based multitasking is light weight.

2. Thread Priorities :

   - Java assigns each thread a priority which defines how that thread should be treated respect to other threads.

   - Thread priorities are integers which specify relative priority of one thread to another.

   - Higher priority thread doesn't mean it runs faster than lower priority thread instead priorities are used to decide when to switch one running thread to the next. This is called **context switch.**

   - Rules that determine when a context switch takes place are simple

     i. A thread can voluntarily relinquish control : This is done by explicitly yielding, sleeping or blocking on pending I/O. In this scenario, all other threads are examined and the highest priority thread which is ready to run is given CPU.

     ii. A thread can be pre-empted by a higher priority thread : In this case a lower priority thread that doesn't yield the processor is simply pre-empted, no matter what its doing, by an higher priority thread. Basically, as soon as a higher priority thread wants to run, it does. This is called **pre-emptive multitasking**.

3. Synchronization :

   A method should be synchronized so that only one thread can use it at one time. Synchronization is very necessary in many cases example when two threads are sharing a data structure like linked list and one thread is updating it while other is reading it then it gives a wrong result. So methods should be synchronized and can be used by one thread at a time and other threads should wait.

4. Thread class and the Runnable interface :

   - To create a new thread a class must either extends **Thread class** or implements **Runnable interface**.

   - Thread class defines several methods, some of those are shown below

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

- Creating a thread by implementing Runnable

```
public
interface Runnable
{
   public abstract void run();
}
```

- One can create thread on any object which implements Runnable.
- Only one method run() needs to be implemented. Inside run() one will define code that constitute the thread. run() can call other methods, use other classes and declare variables just like main thread. The only difference is run() establishes the entry point for another concurrent thread of execution within the program.
- This thread will end when run() returns.
- Example 1 - TCR:

```
package multiThreading;

public class HelloRunnable implements Runnable
{

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}

Output:
Hello from a thread!
```

- Example 2 : Java complete reference

```
package multiThreading;

class ThreadWithRunnable1 implements Runnable
{
Thread t;
```

```
ThreadWithRunnable1()
{
	t = new Thread(this, "Demo thread");
	System.out.println("new thread "+t);
	t.start();
}
public void run()
{
	try
	{
		for(int j=5; j>0; j--)
		{
			System.out.println("child thread "+j);
			Thread.sleep(1000);

		}
	} catch (InterruptedException e)
		{
			System.out.println("child interrupted");
		}
	System.out.println("exiting child thread");
}
}

public class ThreadWithRunnable
{
public static void main(String[] args)
{
	new ThreadWithRunnable1();

		try {
			for(int i=5; i>0; i--)
			{
				System.out.println("main thread "+i);
				Thread.sleep(1000);
			}
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			System.out.println("main thread");
		}
		System.out.println("exiting main thread");
}
}
Output:
new thread Thread[Demo thread,5,main]
main thread 5
child thread 5
main thread 4
child thread 4
main thread 3
child thread 3
child thread 2
main thread 2
child thread 1
main thread 1
exiting main thread
exiting child thread
```

- Example 3 : My program

```
package multiThreading;

public class RunnableThread implements Runnable
{
public static void main(String[] args)
{
	Thread t = new Thread(new RunnableThread());
	t.start();
	//or
	//new Thread(new RunnableThread()).start();
}

public void run()
{
	try {
		for (int i=5; i>0; i--)
		{
			System.out.println("new thread "+i);
			Thread.sleep(1000);
		}
	} catch (InterruptedException e) {
		// TODO Auto-generated catch block
		System.out.println("new thread catch");
	}
	System.out.println("exiting new thread");
}
}
```

```
Output:
new thread 5
new thread 4
new thread 3
new thread 2
new thread 1
exiting new thread
```

Note: Above we can see we can have run method both inside and outside main method.

- Extending Thread Class :
- Example 1

```
package multiThreading;

public class ThreadClassExtend extends Thread
{
public void run()
{
	try {

		for(int i=5; i>0; i--)
		{
			System.out.println("new thread class thread "+i);
			Thread.sleep(1000);
		}
	} catch (InterruptedException e)
	{
		System.out.println("thread class thread");
	}
```

```
        System.out.println("exiting thread class thread");
}

public static void main(String[] args)
{
        Thread t = new ThreadClassExtend();
        t.start();
}
}
Output:
new thread class thread 5
new thread class thread 4
new thread class thread 3
new thread class thread 2
new thread class thread 1
exiting thread class thread
```

- Note : Where run() method is called in above programme ? It is actually called through the start() method. If one even see source code for start() method he cannot find run() method called anywhere in it. Actually inside start() method start0() is called which is actually a native method. So in reality run() method will be called by JVM itself. The result is that two threads are running concurrently: the current thread (which returns from the call to the start method) and the other thread (which executes its run method). I.e. Start() method causes 'this' (i.e. the current thread) to begin execution and JVM will call run method for this thread. If thread was constructed using separate 'Runnable' run object then Runnable object's run() will be called else this method(Thread class 's run()) does nothing and returns. The subclasses of Thread should override this method. Also

- Example 2 :

```
package multiThreading;

class ThreadExtendSeperateClass1 extends Thread
{
public void run()
{
        try {
                for(int j=5; j>0; j--)
                {
                        System.out.println("new thread class thread "+j);
                        Thread.sleep(1000);
                }
        } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                System.out.println("thread class thread");
        }
        System.out.println("exiting thread class");
}
}

public class ThreadExtendSeperateClass
{
public static void main(String[] args)
{
        Thread t = new ThreadExtendSeperateClass1();
        t.start();
```

```
}
}
Output:
new thread class thread 5
new thread class thread 4
new thread class thread 3
new thread class thread 2
new thread class thread 1
exiting thread class
```

5. Chosing an approach

Now we have two ways of creating a thread first by implementing Runnable interface and second by using Thread class. We know by extending Thread class only run() method needed to be overridden and rest of the method doesn't change. So it has been thought to create a Runnable interface and the class in which thread will be created can extend another class too.

6. Main Thread :

- Main thread can be controlled with thread object. To do so one should get reference to it by calling **currentThread()** method. Similarly other Thread methods are applicable too.
- Example

```
package multiThreading;

public class MainThreadClass
{
public static void main(String[] args)
{
	Thread t = Thread.currentThread();
	System.out.println("current thread "+t);

	t.setName("my main");

	System.out.println("main after name change "+t);

}
}
Output:
current thread Thread[main,5,main]
main after name change Thread[my main,5,main]
```

Above in the output `Thread[main,5,main]` represent in order : [Name of thread, priority, name of its group]. Thread group is a data structure that controls the state of a collection of threads as a whole. Next it can be seen that name of thread got changed to 'my main'.

7. Multiple Thread :

Multiple threads can be created in a single program.

Example

```
package multiThreading;

class ManyThreadClass1 implements Runnable
{
```

```
   String name;
   Thread t;

   ManyThreadClass1(String ThreadName)
   {
         name = ThreadName;
         t = new Thread(this, name);
         System.out.println("new thread "+t);
         t.start();
   }

   public void run()
   {
         try {
               for(int i = 5; i > 0; i--)
               {
                     System.out.println(name+" "+i);
                     t.sleep(1000);
               }
         } catch (InterruptedException e)
         {
               System.out.println(name+" exception");
         }
         System.out.println(name+" exiting");
   }
}
public class ManyThreadClass
{
   public static void main(String[] args)
   {
         new ManyThreadClass1("one");
         new ManyThreadClass1("two");
         new ManyThreadClass1("three");

         try {
               Thread.sleep(10000);
               } catch (InterruptedException e) {

                     System.out.println("main exception");
         }
         System.out.println("main exiting");
   }
}
Output:
new thread Thread[one,5,main]
new thread Thread[two,5,main]
new thread Thread[three,5,main]
one 5
two 5
three 5
three 4
two 4
one 4
two 3
three 3
one 3
two 2
three 2
one 2
one 1
two 1
```

```
three 1
two exiting
one exiting
three exiting
main exiting
```

8. isAlive() and join() methods :

- We always want main thread to finish last. In the preceding pgm it is accomplished by calling sleep() within main() with a long delay to ensure that all child threads terminate before main thread. But this is not always the solution because one cannot find out when other threads will terminate. Lets see what will happen if sleep() is not used in the above 'ManyThreadClass' programme and i=3.

  Output:

```
new thread Thread[one,5,main]
new thread Thread[two,5,main]
new thread Thread[three,5,main]
main exiting
one 3
three 3
two 3
three 2
one 2
two 2
three 1
one 1
two 1
three exiting
one exiting
two exiting
```

  Above it can be seen the main is exiting before child threads.

- One way to find out whether particular thread is still running is through **isAlive()** method. This method will return true if the thread on which it is called is still running else it will return false.
- One more method **join()** is very useful in this regards and when it is called on some thread lets say t1 then all the current threads will be paused till t1 will be terminated. Other thread will be terminated only after t1 got terminated.
  - Example 1 :

```
package multiThreading;

class JoinNIsAliveClass1 implements Runnable
{
  String name;
  Thread t;
  JoinNIsAliveClass1(String threadName)
  {
        name = threadName;
        t = new Thread(this, name);
        System.out.println(name+" "+t);
        t.start();
  }

  public void run()
  {
```

```
        try
        {
              for(int i=3; i>0; i--)
              {
                    System.out.println(name+" "+i);
                    Thread.sleep(1000);
              }
        } catch (InterruptedException e) {
              System.out.println(name+" exception");
        }
        System.out.println(name+" exiting");
  }
}

public class JoinNIsAliveClass
{
  public static void main(String[] args)
  {
        JoinNIsAliveClass1 jna1 = new JoinNIsAliveClass1("one");
        JoinNIsAliveClass1 jna2 = new JoinNIsAliveClass1("two");
        JoinNIsAliveClass1 jna3 = new JoinNIsAliveClass1("three");

        System.out.println("Thread one is alive "+jna1.t.isAlive());
        System.out.println("Thread two is alive "+jna2.t.isAlive());
        System.out.println("Thread three is alive "+jna3.t.isAlive());

        //wait for thread to finish
        try {
              System.out.println("waiting for thread to finish");
              jna1.t.join();
              jna2.t.join();
              jna3.t.join();

        } catch (InterruptedException e) {
              System.out.println("main thread interrupted");
        }
        System.out.println("Thread one is alive "+jna1.t.isAlive());
        System.out.println("Thread two is alive "+jna2.t.isAlive());
        System.out.println("Thread three is alive "+jna3.t.isAlive());
        System.out.println("main thread exiting");

  }
}
Output:
one Thread[one,5,main]
two Thread[two,5,main]
three Thread[three,5,main]
Thread one is alive true
Thread two is alive true
Thread three is alive true
waiting for thread to finish
three 3
two 3
one 3
three 2
two 2
one 2
three 1
two 1
one 1
three exiting
```

```
two exiting
one exiting
Thread one is alive false
Thread two is alive false
Thread three is alive false
main thread exiting
```

- Above if one calls only join() on main thread then it will be terminated first before three child threads.

– Example 2

```
package multiThreading;

public class JoinMethodCls implements Runnable{

	String ThreadName;
	Thread t;
	JoinMethodCls(String name)
	{
		ThreadName = name;
		t = new Thread(this, name);
		System.out.println(t);
		try {
			t.start();
			t.join();
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}

	public void run()
	{
		for(int i = 5; i>0; i--)
		{
			try {
				System.out.println(ThreadName+" "+i);
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
		System.out.println("exiting "+ThreadName);
	}

	public static void main(String[] args)
	{
		JoinMethodCls jmc1 = new JoinMethodCls("one");
		JoinMethodCls jmc2 = new JoinMethodCls("two");
		JoinMethodCls jmc3 = new JoinMethodCls("three");

		System.out.println("exiting main");
	}
}
Output:
Thread[one,5,main]
one 5
one 4
```

```
one 3
one 2
one 1
exiting one
Thread[two,5,main]
two 5
two 4
two 3
two 2
two 1
exiting two
Thread[three,5,main]
three 5
three 4
three 3
three 2
three 1
exiting three
exiting main
```

9. Thread priorities :

- One can set the priority of a thread by using method

  final void setPriority(int level)

  The value of 'level' must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently these values are 1 and 10 respectively. The default priority can be specified using **NORMAL_PRIORITY** which is 5. These priorities are defined as static final variables within Thread.
- getPriority() can be used to get the priority of the thread.

10. Synchronization :

- When two or more thread used the same resource then one needs to ensure that resource will be used by one thread at a time. This is done through synchronization. There are two ways through which synchronization works in java. Synchronized methods(both static and non-static) and Synchronized statements(or code blocks both static and non-static).

– Lets say we have two methods inside a class

```
synchronized void f() { /* ... */ }
synchronized void g() { /* ... */ }
```

All objects automatically contain a single lock (also referred to as a *monitor)*. When you call any **synchronized** method, that object is locked and no other **synchronized** method of that object can be called until the first one finishes and releases the lock. For the preceding methods, if **f( )** is called for an object by one task, a different task cannot call **f( )** or **g( )** for the same object until **f( )** is completed and releases the lock. Thus, there is a single lock that is shared by all the **synchronized** methods of a particular object, and this lock can be used to prevent object memory from being written by more than one task at a time.

1. Synchronized methods :

```
package multiThreading;

class SynchWithMethod1
{
	void call(String msg)
	{
		System.out.println("[ "+msg);
		try {
			Thread.sleep(1000);
		} catch (InterruptedException e) {
			System.out.println("SynchWithMethod1 exception");
		}
		System.out.println("]");
	}
}

class SynchWithMethod2 implements Runnable
{
	String str;
	SynchWithMethod1 sm1;
	Thread t;

	public SynchWithMethod2(SynchWithMethod1 sm, String str1)
	{
		str = str1;
		sm1 = sm;
		t = new Thread(this);
		t.start();
	}

	public void run()
	{
		sm1.call(str);
	}
}

public class SynchWithMethod
{
	public static void main(String[] args)
	{
		SynchWithMethod1 smm = new SynchWithMethod1();
		SynchWithMethod2 smm1 = new SynchWithMethod2(smm,
		"hello");
		SynchWithMethod2 smm2 = new SynchWithMethod2(smm,
		"synchranized");
		SynchWithMethod2 smm3 = new SynchWithMethod2(smm,
		"world");
		try
		{
			smm1.t.join();
			smm2.t.join();
			smm3.t.join();
		} catch (InterruptedException e) {
			System.out.println("main exception");
	}
}
```

```
}
Output:
[ hello
[ synchranized
[ world
]
]
]
```

- The expected output was the 'msg' should be printed inside a square bracket and a **sleep()** method occurs in between. but we can see the method **call()** allows execution to switch to another thread when sleep is called. Hence unexpected output occurs.
- Nothing stops three threads from calling a same object, same method at the same time. This is called as **Race Condition**. This means three threads racing each other to complete the method.
- To fix the problem, one should restrict access to the method one thread at a time. It is achieved through **synchronized** key word. If call() method defined in this way

```
synchronized void call(String msg)
{
```

Then output becomes

```
[ hello
]
[ world
]
[ synchranized
]
```

2. Synchronized statements :

- Sometime it may happen that the methods of a class are not synchronized and programme is written by a third party hence there is no access to actual code so that a developer can change it. In such a scenario synchronized statements or synchronized block can be used.
- Synchronized statements must be given an 'object' to synchronized upon. The preceding programme can be rewritten with synchronized statement is as shown below.
- Example :

```
package multiThreading;

class SynchWithStatement1
{
   void call(String msg)
   {
         System.out.println("[ "+msg);
         try {
               Thread.sleep(1000);
         } catch (InterruptedException e) {
               System.out.println("SynchWithStatement1 exception");
         }
         System.out.println("]");
   }
```

```
}

class SynchWithStatement2 implements Runnable
{
   String str;
   SynchWithStatement1 sm1;
   Thread t;

   public SynchWithStatement2(SynchWithStatement1 sm, String str1)
   {
         str = str1;
         sm1 = sm;
         t = new Thread(this);
         t.start();
   }

   public void run()
   {
         synchronized(sm1)
         {
         sm1.call(str);
         }
   }
}

public class SynchWithStatement
{
   public static void main(String[] args)
   {
         SynchWithStatement1 smm = new SynchWithStatement1();
         SynchWithStatement2 smm1 = new SynchWithStatement2(smm,
         "hello");
         SynchWithStatement2 smm2 = new SynchWithStatement2(smm,
         "synchranized");
         SynchWithStatement2 smm3 = new SynchWithStatement2(smm,
         "world");

         try
         {
               smm1.t.join();
               smm2.t.join();
               smm3.t.join();
         } catch (InterruptedException e) {
               System.out.println("main exception");
   }
}
}
```

Output:

```
[ hello
]
[ world
]
[ synchranized
]
```

3. When synch work :

```
package multiThreading;

class SimpleClass
{
   synchronized void method1(String str)
   {
         System.out.println(str+" method1 starts");

         try {
               Thread.sleep(5000);
         } catch (InterruptedException e) {
               System.out.println("method1 interrupted");
         }

         System.out.println(str+" method1 completes");
   }
}

class Class1 implements Runnable
{
   String str1;
   SimpleClass sc1;
   Thread t;
   Class1(SimpleClass sc, String str)
   {
         str1 = str;
         sc1 = sc;
         t = new Thread(this);
         t.start();
   }
   public void run()
   {
         sc1.method1(str1);
   }
}

class Class2 implements Runnable
{
   String str2;
   SimpleClass sc2;
   Thread t;
   Class2(SimpleClass sc, String str)
   {
         str2 = str;
         sc2 = sc;
         t = new Thread(this);
         t.start();
   }
   public void run()
   {
         sc2.method1(str2);
   }
}

public class OneClassTwoObjSynch {

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

```
        SimpleClass sc11 = new SimpleClass();
        SimpleClass sc12 = new SimpleClass();
        SimpleClass sc21 = new SimpleClass();
        SimpleClass sc22 = new SimpleClass();

        //same class, diff instance, same method instance- synch matters
        Class1 c11 = new Class1(sc11, "Class1c11SameMethodInst");
        Class1 c12 = new Class1(sc11, "Class1c12SameMethodInst");

        //same class, diff instance, separate method instance - synch
        doesnt matters
        Class1 c13 = new Class1(sc11, "Class1c13DiffMethodInst");
        Class1 c14 = new Class1(sc12, "Class1c14DiffMethodInst");

        //Diff class, diff instance, Same method instance - synch
        matters
        Class1 c15 = new Class1(sc11, "Class1c15SameMethodInst");
        Class2 c21 = new Class2(sc11, "Class2c21SameMethodInst");

        //Diff class, diff instance, diff method instance - synch doesnt
        matters
        Class1 c16 = new Class1(sc11, "Class1c16DiffMethodInst");
        Class2 c22 = new Class2(sc12, "Class2c22DiffMethodInst");
}

}
```

```
Output:
Class1c11SameMethodInst method1 starts
Class1c14DiffMethodInst method1 starts
Class1c11SameMethodInst method1 completes
Class1c16DiffMethodInst method1 starts
Class1c14DiffMethodInst method1 completes
Class2c22DiffMethodInst method1 starts
Class1c16DiffMethodInst method1 completes
Class1c13DiffMethodInst method1 starts
Class2c22DiffMethodInst method1 completes
Class1c13DiffMethodInst method1 completes
Class2c21SameMethodInst method1 starts
Class2c21SameMethodInst method1 completes
Class1c15SameMethodInst method1 starts
Class1c15SameMethodInst method1 completes
Class1c12SameMethodInst method1 starts
Class1c12SameMethodInst method1 completes
```

i. For clear understanding lets execute two cases at one go and comment out others.

```
1.//same class, diff instance, same method instance - synch matters
  Class1 c11 = new Class1(sc11, "Class1c11SameMethodInst");
  Class1 c12 = new Class1(sc11, "Class1c12SameMethodInst");
  Ouptut:
  Class1c11SameMethodInst method1 starts
  Class1c11SameMethodInst method1 completes
  Class1c12SameMethodInst method1 starts
  Class1c12SameMethodInst method1 completes

2.//same class, diff instance, separate method instance - synch doesnt
  matters
  Class1 c13 = new Class1(sc11, "Class1c13DiffMethodInst");
  Class1 c14 = new Class1(sc12, "Class1c14DiffMethodInst");
  Output:
  Class1c13DiffMethodInst method1 starts
```

```
Class1c14DiffMethodInst method1 starts
Class1c14DiffMethodInst method1 completes
Class1c13DiffMethodInst method1 completes
```

```
3.//Diff class, diff instance, Same method instance - synch matters
  Class1 c15 = new Class1(sc11, "Class1c15SameMethodInst");
  Class2 c21 = new Class2(sc11, "Class2c21SameMethodInst");
  Output:
  Class1c15SameMethodInst method1 starts
  Class1c15SameMethodInst method1 completes
  Class2c21SameMethodInst method1 starts
  Class2c21SameMethodInst method1 completes
```

```
4.//Diff class, diff instance, diff method instance - synch doesnt
  matters
  Class1 c16 = new Class1(sc11, "Class1c16DiffMethodInst");
  Class2 c22 = new Class2(sc12, "Class2c22DiffMethodInst");
  Ouptut:
  Class1c16DiffMethodInst method1 starts
  Class2c22DiffMethodInst method1 starts
  Class1c16DiffMethodInst method1 completes
  Class2c22DiffMethodInst method1 completes
```

ii. This clearly explains Synchronization is only needed when two or more threads are working on the **same object** at a same time. Ex : Two instances of same class c11-12 or two instances of diff class c15-c21 works on same object sc11 through two different threads Synchronization is needed.

iii. When two objects of the same class c13-14 or diff class c16-c22 works on two different instance at a same time sc11, sc12 resp. then synchronization is not needed.

11. Interthread Communication through **wait()** and **notify()** :

- In many occasions we will find that there is a dependency of one thread on another. For example simple producer and consumer classes. There is a int value N where producer writes and consumer reads. We know that thread works mainly according to processors multithreading settings. It may happen that producer may be writing the integer value but consumer is not reading it and when consumer reads it will read only the latest value and haven't read the previous ones. This happens because there is no proper communication between these two threads. The output produced through this is shown below with an example.

- Example :

```
package multiThreading;

class Q
{
 int n;

 synchronized void put(int n)
 {

       this.n = n;
       try {
             Thread.sleep(1000);
       } catch (InterruptedException e) {
             System.out.println("inside put thread");
```

```
        }
        System.out.println("put: "+n);
  }

  synchronized int get()
  {
        System.out.println("get: "+n);
        return n;
  }
}

class Producer implements Runnable
{
  Q q;
  Producer(Q q)
  {
        this.q = q;
        new Thread(this, "producer").start();
  }
  public void run()
  {
        for(int i=0; i<5; i++)
        {
              q.put(i);
        }
  }
}

class Consumer implements Runnable
{
  Q q;
  Consumer(Q q)
  {
        this.q = q;
        new Thread(this, "consumer").start();
  }

  public void run()
  {
        while(q.get()< 4)
        {
              q.get();

        }
  }
}

public class ProblemWithoutWaitNNotify
{
  public static void main(String[] args)
  {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);
  }
}
Output:
put: 0
put: 1
get: 1
get: 1
```

```
get: 1
get: 1
get: 1
get: 1
get: 1
get: 1
put: 2
put: 3
put: 4
get: 4
get: 4
```

```
Note : Above output may very from machine to machine depends on its
multithreading workings.
```

As seen above in the output when 'put' was 0, 'get' was not reading it. But when put became 1 get started reading it many times because put went to sleep. But again when put was 2 and after 3, get was not reading it. In programme we can see the condition while(q.get()<4) has been used but still get:4 got executed. This is not because of while loop was not working but when q.get was 3 get started reading the integer value but meanwhile it got updated with 4 by put.

- wait() and notify() method:

  wait():

```
public final void wait()throws InterruptedException
```

Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object. In other words, this method behaves exactly as if it simply performs the call wait(0).

The current thread must own this object's monitor. The thread releases ownership of this monitor and waits until another thread notifies threads waiting on this object's monitor to wake up either through a call to the notify method or the notifyAll method. The thread then waits until it can re-obtain ownership of the monitor and resumes execution.

As in the one argument version, interrupts and spurious wakeups are possible, and this method should always be used in a loop:

```
synchronized (obj)
{
    while (<condition does not hold>)
        obj.wait();
    ... // Perform action appropriate to condition
}
```

This method should only be called by a thread that is the owner of this object's monitor. See the notify method for a description of the ways in which a thread can become the owner of a monitor.

IllegalMonitorStateException - if the current thread is not the owner of the object's monitor.

InterruptedException - if any thread interrupted the current thread before or while the current thread was waiting for a notification. The *interrupted status* of the current thread is cleared when this exception is thrown.

notify() :

```
public final void notify()
```

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the wait methods.

The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object. The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object.

This method should only be called by a thread that is the owner of this object's monitor. A thread becomes the owner of the object's monitor in one of three ways:

- By executing a synchronized instance method of that object.
- By executing the body of a synchronized statement that synchronizes on the object.
- For objects of type Class, by executing a synchronized static method of that class.

Only one thread at a time can own an object's monitor.

- Threads often have to coordinate their actions. The most common coordination idiom is the *guarded block*. Such a block begins by polling a condition which must be true before the block can proceed. It is resolved by using wait() and notify() or notifyAll() methods. Both of these methods are part of Object class. wait() suspends the current thread and the invocation of wait doesn't return until another thread has issued a notification.
- Note: Always invoke wait inside a loop that tests for the condition being waited for. Don't assume that the interrupt was for the particular condition you were waiting for, or that the condition is still true.
- Shown below in improved version of preceding programme.
- Example 1 :

```
package multiThreading;

class Q1
{
 int n;
 boolean valueSet = false;
```

```
 synchronized void put(int n)
 {
       while(valueSet)
             try {
                   wait();
             } catch (InterruptedException e) {
                   System.out.println("inside put wait");
             }
       this.n = n;
       System.out.println("put: "+n);
       valueSet = true;
       notify();
 }

 synchronized int get()
 {
       while(!valueSet)
       try {
             wait();
       } catch (InterruptedException e) {
             System.out.println("inside get wait");
       }
       System.out.println("get: "+n);
       valueSet = false;
       notify();
       return n;
 }
}

class Producer1 implements Runnable
{
 Q1 q;
 Producer1(Q1 q)
 {
       this.q = q;
       new Thread(this, "producer1").start();
 }
 public void run()
 {
       for(int i=0; i<5; i++)
       {
             q.put(i);
       }
 }
}

class Consumer1 implements Runnable
{
 Q1 q;
 Consumer1(Q1 q)
 {
       this.q = q;
       new Thread(this, "consumer1").start();
 }

 public void run()
 {
       while(q.get()< 4)
       {
             q.get();
```

```
        }
  }
}

public class SolutionWithWaitNNotify
{
  public static void main(String[] args)
  {
        Q1 q = new Q1();
        new Producer1(q);
        new Consumer1(q);
  }
}
Output:
put: 0
get: 0
put: 1
get: 1
put: 2
get: 2
put: 3
get: 3
put: 4
get: 4
```

- It is very necessary to understand how the above program is working. When important point is when put() or get(), get into while loop and execute wait() then that thread got suspended and all instructions coming after while loop will not going to execute. Similarly when wait() is not invoked then only instructions written after while loop will be executed.

– Example 2:

```
package testProg;

public class ThreadA {
    public static void main(String[] args){
        ThreadB b = new ThreadB();
        b.start();

        synchronized(b){
           try
           {
                System.out.println("Waiting for b to complete...");
                b.wait();
           }catch(InterruptedException e)
           {
                e.printStackTrace();
           }
             System.out.println("Total is: " + b.total);
        }
    }
}

class ThreadB extends Thread{
    int total;
    @Override
    public void run(){
        synchronized(this){
```

```
                for(int i=0; i<100 ; i++){
                    total += i;
                }
                notify();
            }
        }
    }
    Output:
    Waiting for b to complete...
    Total is: 4950
```

– If One removes wait() method from above programme then output would be

```
Output:
Waiting for b to complete...
Total is: 0
```

11. DeadLock :

- Deadlock occurs when two threads have a circular dependency on a pair of synchronized objects.
- Example:

```
package multiThreading;

class A
{
   synchronized void foo(B b)
   {
         String name = Thread.currentThread().getName();
         System.out.println(name+" entered A.foo");

         try {
               Thread.sleep(1000);
         } catch (InterruptedException e) {
               System.out.println("A Interrupted");
         }
         System.out.println(name+" trying to call B's last");
         b.last();
   }

   synchronized void last()
   {
         System.out.println("inside A.last");
   }
}

class B
{
   synchronized void bar(A a)
   {
         String name = Thread.currentThread().getName();
         System.out.println(name+" entered B.bar");
         try {
               Thread.sleep(1000);
         } catch (InterruptedException e) {
               System.out.println("B Interrupted");
         }
         System.out.println(name+" trying to call A's last");
         a.last();
```

```
    }

    synchronized void last()
    {
          System.out.println("inside B.last");
    }
}
public class DeadLockPgm implements Runnable
{
    A a = new A();
    B b = new B();

    DeadLockPgm()
    {
          Thread.currentThread().setName("MainThread");
          Thread t = new Thread(this, "Racing Thread");
          t.start();
          a.foo(b);
          System.out.println("back in main thread");
    }
    public void run()
    {
          b.bar(a);
          System.out.println("back in other thread");
    }

    public static void main(String[] args)
    {
          new DeadLockPgm();
    }
}
Output:
MainThread entered A.foo
Racing Thread entered B.bar
Racing Thread trying to call A's last
MainThread trying to call B's last
```

- Above programme never ends and A's last and B's last will be never called because of deadlock.

12. Suspending, Resuming and Stopping Threads :

- Sometime it is necessary to suspended(or paused) a thread and suspended thread can be resumed at any time. In **java 1.0** both of these tasks are achieved through **suspend()** and **resume()** method defined in thread class. Now both of these methods have been deprecated(will be scratched below in the pgm). But it is also necessary to know them as one may need to work on legacy classes for some old codes. Example for them is shown below.
- Example :

```
package multiThreading;

class SuspendResumeClass1 implements Runnable
{
    String str;
    Thread t;
    SuspendResumeClass1(String name)
    {
          str = name;
        t = new Thread(this, str);
```

```
        System.out.println("new thread :"+t);
        t.start();
    }
    public void run()
    {
          try {
                for(int i = 15; i>0 ; i--)
                {
                      System.out.println(str+" :"+i);
                      Thread.sleep(200);
                }
          } catch (InterruptedException e) {

                System.out.println(str+" interrupted");
          }
          System.out.println(str+" exiting");
    }
}

public class SuspendResumeClass
{
    public static void main(String[] args)
    {
          SuspendResumeClass1 src1 = new SuspendResumeClass1("one");
          SuspendResumeClass1 src2 = new SuspendResumeClass1("two");

          try {
                Thread.sleep(1000);
                src1.t.suspend();
                System.out.println("Suspending thread one");
                Thread.sleep(1000);
                src1.t.resume();
                System.out.println("resuming thread one");
                src2.t.suspend();
                System.out.println("Suspending thread two");
                Thread.sleep(1000);
                src2.t.resume();
                System.out.println("resuming thread two");

          } catch (InterruptedException e) {
                System.out.println("main interrupted");
          }
          try {
                System.out.println("wating for thread to finish");
                src1.t.join();
                src2.t.join();
          } catch (InterruptedException e) {
                System.out.println("main interrupted");
          }
          System.out.println("exiting main");
    }
}
Output:
new thread :Thread[one,5,main]
new thread :Thread[two,5,main]
one :15
two :15
one :14
two :14
one :13
two :13
```

```
one :12
two :12
one :11
two :11
one :10
Suspending thread one
two :10
two :9
two :8
two :7
two :6
resuming thread one
Suspending thread two
one :9
one :8
one :7
one :6
one :5
one :4
two :5
resuming thread two
wating for thread to finish
two :4
one :3
one :2
two :3
one :1
two :2
two :1
one exiting
two exiting
exiting main
```

- A thread can be stopped using method stop() and once stopped then thread cannot be resumed again.
- Example 2:

```
public static Object cacheLock = new Object();
  public static Object tableLock = new Object();
  ...
  public void oneMethod() {
    synchronized (cacheLock) {
      synchronized (tableLock) {
        doSomething();
      }
    }
  }
  public void anotherMethod() {
    synchronized (tableLock) {
      synchronized (cacheLock) {
        doSomethingElse();
      }
    }
  }
```

Now, imagine that thread A calls oneMethod() while thread B simultaneously callsanotherMethod(). Imagine further that thread A acquires the lock on cacheLock, and, at the same time, thread B acquires the lock on tableLock. Now the threads are deadlocked: neither thread will give up its lock until it acquires the other lock, but neither will be able to acquire the other lock until the other thread gives it up. When a Java program deadlocks, the deadlocking threads simply wait forever. While other threads might continue running, you will eventually have to kill the program, restart it, and hope that it doesn't deadlock again.

- After Java 2.0 :

  After java 2.0 it has been found that method suspend() can cause serious system failures. For example if a thread is accessing critical synchronized data structure and if it has been suspended then the lock for the method dont get released and other threads waiting for the methods get deadlocked. Method resume() has been deprecated because it is only useful with suspend().
- Method stop() too has been deprecated because it is too can cause problems. Lets say for example a thread is writing something to a critical synchronized data structure and in between it has been stopped. It will make it unlock but the other thread who are going to accessing the same data structure get corrupted version of it because of half work done by the previous thread.
- All suspend(), resume() and stop() has been deprecated in java 2.0.
- In place of them Object methods wait() and notify() will be used for the same purpose. Below programme explains how.

```
package multiThreading;

class SuspendResumeWithWaitNNotify1 implements Runnable
{
   String str;
   Thread t;
   boolean SuspendFlag;
   SuspendResumeWithWaitNNotify1(String name)
   {
         str = name;
       t = new Thread(this, str);
       System.out.println("new thread :"+t);
       SuspendFlag = false;
       t.start();
   }

   public void run()
   {
         try {
               for(int i = 15; i>0 ; i--)
               {
                     System.out.println(str+" :"+i);
                     Thread.sleep(200);
                     synchronized(this)
                     {
                           while(SuspendFlag)
                           {
                                 wait();
                           }
                     }
               }
         } catch (InterruptedException e) {

               System.out.println(str+" interrupted");
         }
         System.out.println(str+" exiting");
   }

   synchronized void mySuspend()
```

```
   {
         SuspendFlag = true;
   }

   synchronized void myResume()
   {
         SuspendFlag = false;
         notify();
   }
}

public class SuspendResumeWithWaitNNotify
{
   public static void main(String[] args)
   {
         SuspendResumeWithWaitNNotify1 src1 = new
SuspendResumeWithWaitNNotify1("one");
         SuspendResumeWithWaitNNotify1 src2 = new
SuspendResumeWithWaitNNotify1("two");

         try {
               Thread.sleep(1000);
               src1.mySuspend();
               System.out.println("Suspending thread one");
               Thread.sleep(1000);
               src1.myResume();
               System.out.println("resuming thread one");
               src2.mySuspend();
               System.out.println("Suspending thread two");
               Thread.sleep(1000);
               src2.myResume();
               System.out.println("resuming thread two");
               } catch (InterruptedException e) {
               System.out.println("main interrupted");
         }
         try {
               System.out.println("wating for thread to finish");
               src1.t.join();
               src2.t.join();
         } catch (InterruptedException e) {
               System.out.println("main interrupted");
         }
         System.out.println("exiting main");
   }
}
Output:
new thread :Thread[one,5,main]
new thread :Thread[two,5,main]
one :15
two :15
one :14
two :14
one :13
two :13
one :12
two :12
one :11
two :11
one :10
Suspending thread one
two :10
```

```
two :9
two :8
two :7
two :6
resuming thread one
Suspending thread two
one :9
one :8
one :7
one :6
one :5
resuming thread two
wating for thread to finish
two :5
one :4
two :4
one :3
two :3
one :2
two :2
one :1
two :1
one exiting
two exiting
exiting main
```

13. Obtaining Thread State's :

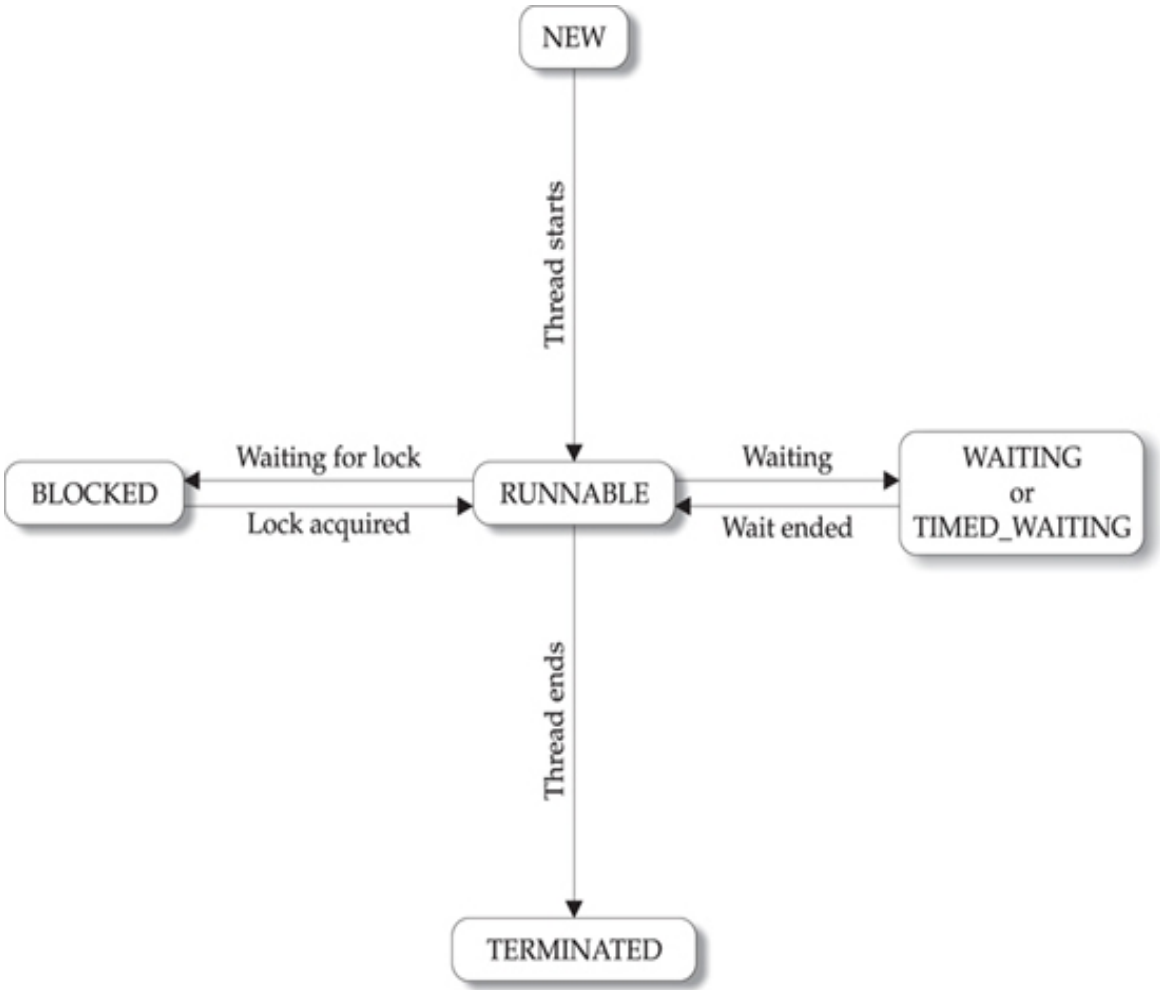
- A thread can exist in different States. One can get the current state of thread using **getState()** method defined by Thread. It is show here

  Thread.State getState()

- It returns a value of type Thread.State which indicate state of thread at the time the call was made.

  State is an enumeration defined by Thread. Below are values returned by getState().

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called **sleep( )**. This state is also entered when a timeout version of **wait( )** or **join( )** is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of **wait( )** or **join( )**.



- Below example shows state of thread at different stages.
- Example :

```
package multiThreading;

class MethodClass1
{
   synchronized void method1()
   {
         System.out.println("inside method class");
   }
}

class ThreadStateClass1 implements Runnable
{
   MethodClass1 mc;
   String str;
   Thread t;
```

```
   ThreadStateClass1(MethodClass1 mc1, String name)
   {
         mc = mc1;
         str = name;
         t = new Thread(this, str);
         t.start();
         System.out.println("after start "+t.getName()+"
         :"+t.getState());
         System.out.println("after start "+t.getName()
         +" :"+t.getState());
   }
   public void run()
   {

         mc.method1();
         try {
               Thread.sleep(1000);
             } catch (InterruptedException e) {
                         System.out.println(str+" interrupted");
             }
         System.out.println("after run "+t.getName()+" :"+t.getState());
         System.out.println("after run "+t.getName()+" :"+t.getState());

   }
}
public class ThreadStateClass
{
   public static void main(String[] args)
   {
         MethodClass1 mc = new MethodClass1();
         ThreadStateClass1 tsc1 = new ThreadStateClass1(mc, "one");
         ThreadStateClass1 tsc2 = new ThreadStateClass1(mc, "two");

         System.out.println("from main "+tsc1.t.getName()
         +" :"+tsc1.t.getState());
         System.out.println("from main "+tsc2.t.getName()
         +" :"+tsc2.t.getState());

         try {
               tsc1.t.join();
               tsc2.t.join();
               System.out.println("after join "+tsc1.t.getName()
               +" :"+tsc1.t.getState());
               System.out.println("after join "+tsc2.t.getName()
               +" :"+tsc2.t.getState());
         } catch (InterruptedException e) {
               System.out.println("main interrupted");
         }
         System.out.println("end of main "+tsc1.t.getName()
         +" :"+tsc1.t.getState());
         System.out.println("end of main "+tsc2.t.getName()
         +" :"+tsc2.t.getState());
   }

}
Output:
after start one :RUNNABLE
inside method class
after start one :RUNNABLE
```

```
after start two :RUNNABLE
after start two :RUNNABLE
from main one :TIMED_WAITING
from main two :RUNNABLE
inside method class
after run two :RUNNABLE
after run one :RUNNABLE
after run two :RUNNABLE
after run one :RUNNABLE
after join one :TERMINATED
after join two :TERMINATED
end of main one :TERMINATED
end of main two :TERMINATED
```

Chapter 15. Exceptional Handling

1) Introduction :

➔ When an exception condition arises in a method, an object representing that exception will be created and thrown in the method.

➔ That method may chose to handle the exception itself or pass it on. At some point it will be caught and processed.

➔ Exception could be generated by Java run time system or by programmers code.

➔ Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.

➔ Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.

➔ Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

➔ All exception types are subclass of built in class **Throwable**. Hence Throwable is at the top of exception class hierarchy. Below throwable there are two subclasses which partition exceptions in two distinct branches. One is **Exception** which is made for exceptional condition which user programs should catch. One has to extend this class to create custom exceptions.

```
public class Exception extends Throwable
```

➔ There is important subclass of Exception which is **RunTimeException**. Exceptions of these types are automatically defined for the programs one write(like ArithmeticException).

➔ Other branch is topped by Error.

```
public class Error extends Throwable
```

➔ Exceptions of type Error are not expected to be caught under normal circumstances by one's program. They are used by Java run time system to indicate errors caused by Java run time environment like stack overflow.

➔ Example :

```
package ExceptionHandling;
public class UncatchedExceptionCls
{
      public static void main(String[] args)
      {
                  int d = 0;
```

```
                int a = 42 / d;

        }
}
Output:
java.lang.ArithmeticException: / by zero
at ExceptionHandling.UncatchedExceptionCls.main(UncatchedExceptionCls.java:10)
```

➔ In the above example When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **UncatchedExceptionCls** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system. (The above output is got after putting code in try/catch pair just for clarity).

➔ The stack trace will always show the sequence of method invocations that led up to the error. In the example below same exception is caught but in a method separate from **main( )**:

```
class Exc1 {
static void subroutine() {
int d = 0;
int a = 10 / d;
}
public static void main(String args[]) {
Exc1.subroutine();
}
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:4)
at Exc1.main(Exc1.java:7)
```

2) <u>Try and Catch</u> :

➔ Its better to handle Exceptions by programmer himself because of two reasons. One, its allows to fix the error and two, its prevents the programme from automatic termination.

Example:

```
package ExceptionHandling;

    public class TryCatchSimple
    {
      public static void main(String[] args)
      {

      try {
                int d = 0;
                int a = 42 / d;
                System.out.println("this wont be printed");
            }
```

```
            catch (ArithmeticException e)
            {
                  System.out.println("Division by zero");
            }
            System.out.println("after catch statement");
      }
}
Output:
Division by zero
after catch statement
```

➔ Throwable class overrides toString() method so that it returns a string containing a description of the exception.

2.1 Multiple Catch clauses :

In some cases, a piece of code can generate more than one type of exceptions. In such cases one can use multiple case statements. When exception occurs catch statements are checked one by one and the one which matches gets executed and rest catch statements will be bypassed.

➔ Example :

```
package ExceptionHandling;

public class MultipleCatchCls
{
      public static void main(String[] args)
      {
            //String [] str = {"Jayant"};
            //args = str;
            try {
                  int a = args.length;
                  int div = 40/a;
                  int [] intArr = {1};
                  intArr[23] =55;
            } catch (ArithmeticException e) {
                  System.out.println("divide by zero :"+e);
            }catch(ArrayIndexOutOfBoundsException e)
            {
                  System.out.println("Array outofBounds :"+e);
            }
            System.out.println("After try/catch");
      }
}
Output:
divide by zero :java.lang.ArithmeticException: / by zero
After try/catch
```

➔ If we remove the comments (//) in the above program then output will be

```
Array outofBounds :java.lang.ArrayIndexOutOfBoundsException: 23
After try/catch
```

Note: We can give command line arguments in eclipse through 'Right' clicking on .class file → Run As → Run Configuration → Arguments(tab) → Program arguments.

2.2 Nested try statements :

➔ Try statements can be nested. Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted. If no **catch** statement matches, then the Java run-time system will handle the exception.

➔ Example 1:

```
package ExceptionHandling;

public class NestedTryCls
{

	public static void main(String[] args)
	{

			try {
				int a = args.length;

				int b = 23/a;

				try
				{
				if(a==1)
				a = a/(a-a);


				if(a==2)
				{
					int [] c = {1};
					c[49] = 12;
				}
				} catch (ArrayIndexOutOfBoundsException e)
					{
					System.out.println("Array out of bounds :"+e);
					}
				} catch (ArithmeticException e)
					{
					System.out.println("Arithmetic excptn :"+e );
				}
	}
}
Output:

C:>java NestedTryCls
Arithmetic excptn :java.lang.ArithmeticException: / by zero

C:>java NestedTryCls Hello
Arithmetic excptn :java.lang.ArithmeticException: / by zero

C:>java NestedTryCls Hello World
Array out of bounds :java.lang.ArrayIndexOutOfBoundsException: 49
```

➔ Above we can see when a = 1, the inner try block cannot catch divide by zero exception. Hence it will be passed to outer try block, where it is handled. When a=2 inner try block will catch ArrayIndexOutOfBoundsException.

➔ Example 2 : (How try nesting works between diff classes or methods)

```
package exceptionHandling;

class NestedTryCls12
{
	void method(int i)
	{
		try {
			System.out.println("inside method()");
			i = i/(i-i);
			int [] intArr = {1};
			intArr[22] = 45;
		} catch (ArrayIndexOutOfBoundsException e)
		{
			System.out.println("Array IOB : "+e);
		}
	}
}

public class NestedTryCls1
{
	public static void main(String[] args)
	{
		NestedTryCls12 nt12 = new NestedTryCls12();
		try {

			int a = args.length;
			int b = 32/a;

			nt12.method(a);

		} catch (ArithmeticException e) {
			System.out.println("Arithmetic expt : "+e);
		}
	}
}
Output:

C:>java NestedTryCls1
Arithmetic expt : java.lang.ArithmeticException: / by zero

C:>java NestedTryCls1 Hello
inside method()
Arithmetic expt : java.lang.ArithmeticException: / by zero

C:>java NestedTryCls1 Hello World
Array IOB : java.lang.ArrayIndexOutOfBoundsException: 22
```

3. 'throw' :

➔ It is possible to throw an Exception explicitly using 'throw' statement as shown below

throw *throwableInstance*;

➔ Here throwableInstance is a object of type Throwable or subclass of Throwable. There are two ways one can get a Throwable object. Either by using a parameter in catch clause or creating one with new operator.

➔ No statement will be executed after throw statement. Control will be passed to the nearest try/catch block. If no matching exception found there then it will go to outer try block and so on. If there is nothing to catch this Exception then Java default exception handler will take control.

➔ Example :

```
package exceptionHandling;

class throwDemo1
{
	void method()
	{
		try {
			throw new NullPointerException("demo");
		} catch (NullPointerException e)
		{
			System.out.println("caught inside method");
			throw e;
		}
	}
}

public class throwDemo {

	public static void main(String[] args)
	{
		try {
			throwDemo1 td1 = new throwDemo1();
			td1.method();
		} catch (NullPointerException e) {
			System.out.println("Recaught "+e);
		}
	}
}
Output:
caught inside method
Recaught java.lang.NullPointerException: demo
```

➔ In the line

```
new NullPointerException("demo");
```

We can see how Exception creation takes place. Many of Java's built-in exception type have two constructors. One is without String and one is with String. The String ('demo' in this case) is used to describe the exception and get displayed when Exception object has been sent as argument to print() or println() methods.

4. Throws :

➔ If any method is capable of throwing certain Exception then it should specify this behavior so that caller of that method can guard itself against it. It is done by including **throws** clause in the method's declarations.

➔ throws lists all the exceptions that a method could throw except Error or RunTimeException or any of their subclasses. If it dont then compile time error will occur.

➔ General form of method declaration with throws clause is as shown below

type method-name(*parameter-list*) throws *exception-list*

{
// body of method
}

Here exception-list is comma separated list of Exceptions that method can throw.

➔ Example :

```
package exceptionHandling;

public class ThrowsCls {

	static void throwsMethod()
	{
		//throw new NullPointerException("nullpt");//This was fine
		System.out.println("Inside throwsMethod()");
		throw new IllegalAccessException("throw");//ERROR
	}

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

The above code wont compile because of 'Unhandled exception type IllegalAccessException' needs to 'Add throws declaration'. To compile and run it properly one needs to add throws declaration to the method throwsMethod() and main() method must define try/catch block to catch this Exception as shown below.

```
package exceptionHandling;

public class ThrowsCls {

	static void throwsMethod() throws IllegalAccessException
	{
		System.out.println("Inside throwsMethod()");
		throw new IllegalAccessException("throwsDemo");
	}

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

```
	{
		try {
			throwsMethod();
		} catch (IllegalAccessException e) {
			System.out.println("caught "+e);
		}

	}

}
Output:
Inside throwsMethod()
caught java.lang.IllegalAccessException: throwsDemo
```

5. finally :

➔ When Exception occurs program execution takes abrupt non-linear path that alters the normal flow. Let in case if a method opens a file and reading through it and completes immaturely because of Exception and kept the file open. In such cases keyword finally comes handy.

➔ finally block comes after try/catch block and executes regarding of Exception occurred or not. It also executes even before method's return statement.

➔ Each try block should have either one catch block or finally block.

➔ Example :

```
package exceptionHandling;

public class FinallyDemoCls {
	static void finMeth1()
	{
		try {
			System.out.println("Inside finMeth1()");
			throw new RuntimeException("RunTime");
		} finally
		{
			System.out.println("finMeth1's finally");
		}
	}

	static void finMeth2()
	{
		try {
			System.out.println("Inside finMeth2()");
			return;
		} finally{
			System.out.println("finMeth2's finally");
		}
	}

	static void finMeth3()
	{
		try {
			System.out.println("Inside finMeth3()");
			} finally{
```

```
                System.out.println("finMeth3's finally");
            }
        }

        public static void main(String[] args)
        {
            try {
                finMeth1();
            } catch (Exception e) {
                System.out.println("Exception is caught");
            }
            finMeth2();
            finMeth3();
        }
}
Output:
Inside finMeth1()
finMeth1's finally
Exception is caught
Inside finMeth2()
finMeth2's finally
Inside finMeth3()
finMeth3's finally
```

6. <u>Java's Built-in Exceptions</u>:

➔ Java has **java.lang** package which defines several exception classes. The most general Exceptions are subclasses of RuntimeException. Java has ***unchecked*** Exceptions because compiler doesn't checks these Exceptions to see if method handles or throws this exceptions. Then there are ***checked*** exceptions and if any method could throw one or more of them then it should be included in the methods throws list. Java also defines various other Exceptions related to its class and libraries.

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

TABLE 10-1 Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

TABLE 10-2 Java's Checked Exceptions Defined in **java.lang**

7. <u>Creating your own Exceptions</u> :

➔ Creating your own Exception is very easy, for that one should extend class Exception. Exception class is subclass of Throwable and just inherit all its methods(don't override any). Exception class found in the java package has been shown below.

```
package java.lang;


public class Exception
  extends Throwable
{
  static final long serialVersionUID = -3387516993124229948L;

  public Exception() {}

  public Exception(String paramString)
  {
    super(paramString);
  }

  public Exception(String paramString, Throwable paramThrowable)
  {
    super(paramString, paramThrowable);
  }

  public Exception(Throwable paramThrowable)
  {
    super(paramThrowable);
  }

  protected Exception(String paramString, Throwable paramThrowable, boolean
paramBoolean1, boolean paramBoolean2)
  {
    super(paramString, paramThrowable, paramBoolean1, paramBoolean2);
  }
}
```

Above we can see Exception class don't define any new method neither override any. The methods defined in class Throwable are shown below.

Method	Description
Throwable fillInStackTrace( )	Returns a **Throwable** object that contains a completed stack trace. This object can be rethrown.
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time, as an array of **StackTraceElement**. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The **StackTraceElement** class gives your program access to information about each element in the trace, such as its method name.
Throwable initCause(Throwable *causeExc*)	Associates *causeExc* with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream *stream*)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter *stream*)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement *elements*[ ])	Sets the stack trace to the elements passed in *elements*. This method is for specialized applications, not normal use.
String toString( )	Returns a **String** object containing a description of the exception. This method is called by **println( )** when outputting a **Throwable** object.

TABLE 10-3 The Methods Defined by **Throwable**

➔ Example :

```
package exceptionHandling;

class CustomException extends Exception
{
	int a;

	CustomException(int i)
	{
		a = i;
	}

	/*public String toString()
```

```
	{
		return "CustomException["+a+"]";
	}*/
}

public class CustomException1
{
	static void compute(int j) throws CustomException
	{
		System.out.println("compute("+j+")");
		if(j>10)
			throw new CustomException(j);
		System.out.println("Normal exit");
	}

	public static void main(String[] args)
	{
		try {
			compute(1);
			compute(20);
		} catch (CustomException e) {
			System.out.println("Caught "+e);
		}
	}
}
Output:
compute(1)
Normal exit
compute(20)
Caught exceptionHandling.CustomException
```

➔ If we comment out toString() method in the above program then output will be

```
Output:
compute(1)
Normal exit
compute(20)
Caught CustomException[20]
```

8. <u>Chained Exceptions</u> :

➔ Sometimes it happens that an ArithmeticException has occurred due to divide by zero but in reality that error has caused due to I/O error occurred. Hence its better that caller of the method should know that underlying cause is I/O error. This is achieved through chained Exceptions which has been added in Java 1.4.

➔ For chained exceptions two constructors have been added to the Throwable class. They are as shown below.

```
Throwable(Throwable causeExc)
Throwable(String msg, Throwable causeExc)
```

Here *causeExc* is a Exception which has causes the current exception. In the second constructor along with the causing exception one can also give description of the current exception. These two constructors also been

added to **Error**, **Exception** and **RuntimeException** classes.

➔ The chained exception methods added to Throwable class are shown below.

```
Throwable getCause( )
Throwable initCause(Throwable causeExc)
```

The getCause() method returns the Exception which underlies the current exception. If there is no underlying exception then null will be returned. The initCause() method associates causeExc with the current exception. The method intiCause() could be only called once for each Exception Object.

➔ Example :

```
package exceptionHandling;

//Demonstrate exception chaining.
class ChainExcDemo
	{
		static void demoproc()
		{
			//create an exception
			NullPointerException e = new NullPointerException("top layer");
			//add a cause
			e.initCause(new ArithmeticException("cause"));
			throw e;
	}

	public static void main(String args[])
	{
		try {
			demoproc();
			} catch(NullPointerException e) {
				//display top level exception
				System.out.println("Caught: " + e);
				//display cause exception
				System.out.println("Original cause: " + e.getCause());
			}
	}
}
Output:
Caught: java.lang.NullPointerException: top layer
Original cause: java.lang.ArithmeticException: cause
```

Chapter 16. Arrays Basics

Topic Covered

1. Declaring, Constructing and Initializing an array.
2. Anonymous array declaration.
3. One dimentional array n two dimentional arrays.
4. NullPointerException and ArrayOutOfBoundsException.
5. Array element assignments for both Primitives and Objects(Objects, subclass objects and Interfaces).
6. Array of Interfaces.
7. Array reference assignments.

---

1. Arrays are objects in java that stores multiple variables of the same type. Arrays can hold either primitives or object references.

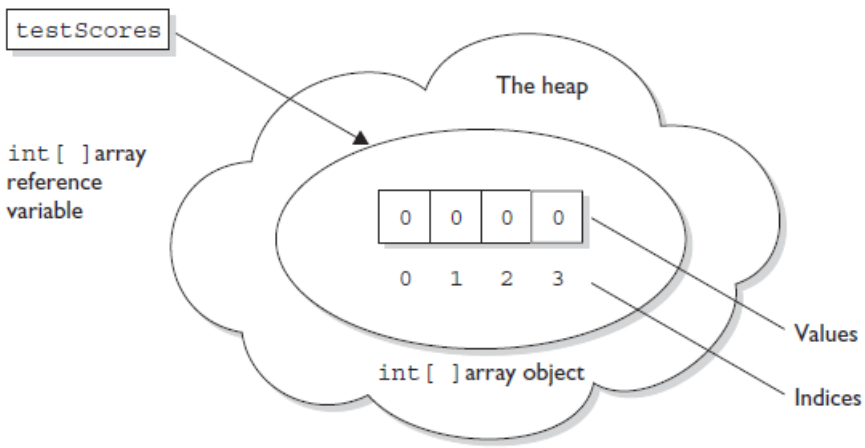
2. Constructing and initializing an array :

```
int [] testScores;//line one

testScores = new int[5];//line two
```

The above can also be written in one line as

```
int [] testScores = new int[5];
```

- Above left part called as declaring an array and right part is called constructing an array i.e. creating an Array object on the heap by doing a new on array type. While declaring an array wont make compiler to allot memory space. It only after constructing or calling new on array type memory is allocated.
- Above testScores is called identifier and int[] is array type i.e. its an array of type int. Arrays must be given a size at the time when they are constructed as JVM needs to allocate space for them. They cannot contract or expand dynamically as collections do. Hence their size is fixed.



3. Lets create array of objects.

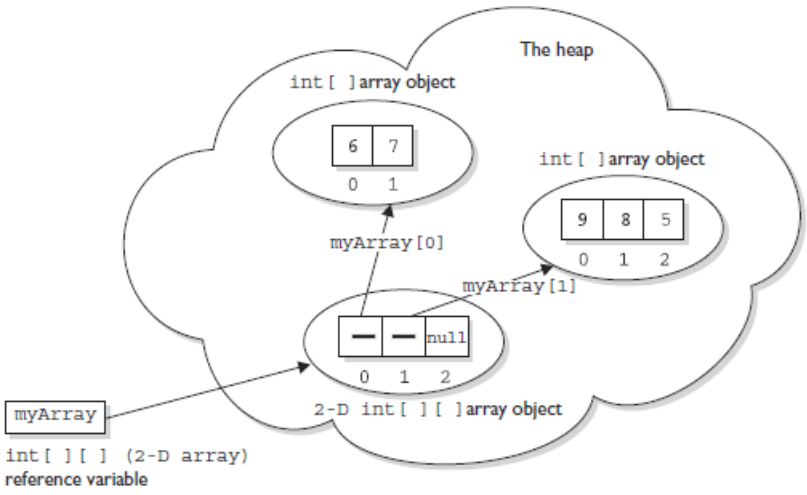
```
Thread[] threads = new Thread[5]; // no Thread objects created!
                                  // one Thread array created
```

Remember through above code thread constructor is not being called. It not creating a thread instance rather Thread array object, so there are no actual thread objects.

4. Constructing an multidimentional array :

```
int[][] myArray = new int[3][];
```

Above code shows how multidimensional array is defined. Multidimensional array is simply an array of arrays. In the above example 'myArray' is an array of arrays of type int. Size defined in the first bracket is enough for JVM to know the size needed to define for this array.



Picture demonstrates the result of the following code:

```
int[ ][ ] myArray = new int[3][ ];
myArray[0] = new int[2];
myArray[0][0] = 6;
myArray[0][1] = 7;
myArray[1] = new int[3];
myArray[1][0] = 9;
myArray[1][1] = 8;
myArray[1][2] = 5;
```

5. Initializing an Array :

- Initializing an array means putting things into it. These things in the arrays are called arrays elements. They are either primitives or object references. If those object references are not

pointing to any objects then accessing then through (.) operator throws **NullPointerException**.

- Array elements are accessed through index number which starts with zero (0) to (length of array – 1). Ex. for array of size 10 index number will run from 0 to 9.
- When someone tries to access an out-of-the-range array index then **ArrayIndexOutOfBoundsException** will be thrown.
- Ex:

```
public class SimpleArray {

public static void main(String[] args)
 {
        int [] intArr = new int[3];
        intArr[0] = 1;
        intArr[1] = 2;
        intArr[2] = 3;
        System.out.println(intArr[0]);
        System.out.println(intArr[1]);
        System.out.println(intArr[2]);
 }
}
```

```
Output:
1
2
3
```

- Array could also be initialized in a loop as shown below

```
public class SimpleArray {

 public static void main(String[] args)
 {

        int [] intArr = new int[3];
        //initialize array intArr
        for(int i=0;i<intArr.length;i++)
              intArr[i] = i+1;
        //print elements of intArr
        for(int j=0;j<intArr.length;j++)
              System.out.println(intArr[j]);
 }
}
```

```
Output:
1
2
3
```

Also for objects, we can loop as shown below.

```
Dog[] myDogs = new Dog[6]; // creates an array of 6 Dog references
for(int x = 0; x < myDogs.length; x++) {
myDogs[x] = new Dog(); // assign a new Dog to index position x
}
```

– When Array elements are initialized then they are set to their default values.

Example:

```
package OverloadingnArrays;

public class ArrayExceptionCls {

	public static void main(String[] args)
	{
		try
		{
			int [] arr = new int[6];
			arr[0] = 0;
			arr[1] = 1;
			arr[4] = 4;
			arr[5] = 5;
		//	arr[6] = 6;//ERROR

			System.out.println("Array Contains");
			for(int i = 0; i<6; i++)
			{

				System.out.println(arr[i]);
			}

		} catch (Exception e)
		{
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
	Output:
	Array Contains
	0
	1
	0
	0
	4
	5
```

– Example for ArrayIndexOutOfBoundsException and NullP

– Example 1: ArrayIndexOutOfBoundsException

```
package OverloadingnArrays;

public class ArrayExceptionCls {

	public static void main(String[] args)
	{
		try
		{
			int [] arr = new int[6];
			arr[0] = 0;
			arr[1] = 1;
			arr[4] = 4;
			arr[5] = 5;
			arr[6] = 6;//ERROR

		} catch (Exception e)
		{
			// TODO Auto-generated catch block
```

```
                e.printStackTrace();
            }
        }
}
Output:
java.lang.ArrayIndexOutOfBoundsException: 6
        at
OverloadingnArrays.ArrayExceptionCls.main(ArrayExceptionCls.java:14)
```

– Also if one tries to access 'arr' array elements through for loop as shown below

```
for(int i = 0; i<=6; i++)
```

then also it will throw ArrayIndexOutOfBoundsException because of '=' in the for loop tries to access arr[6].

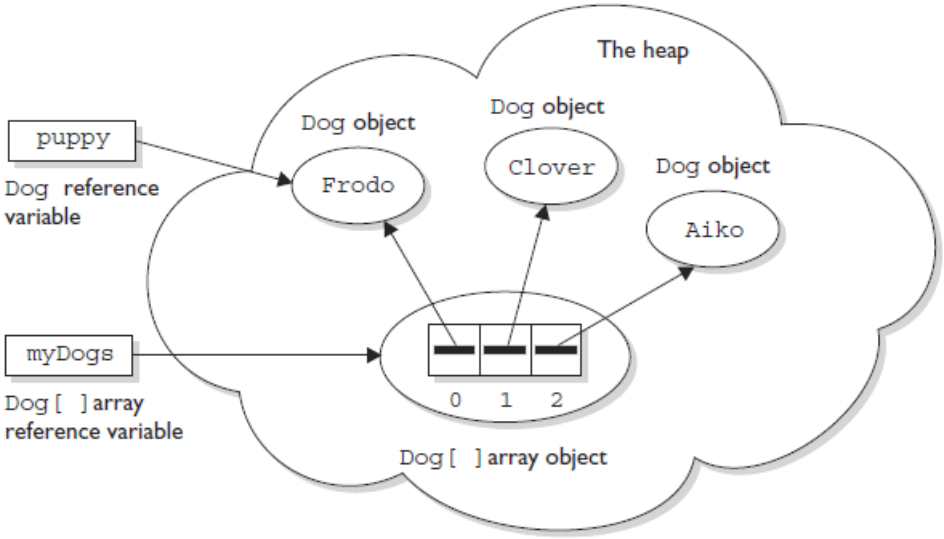
– Example 2 : NullPointerException

```
package OverloadingnArrays;

class A
{
        int a = 10;
}

public class ArrayNullPointerCls {

        public static void main(String[] args)
        {
                A a1 = new A();
                A a2 = new A();
                A a3 = new A();

                A [] objArr = new A[4];
                        //{a1, a2, a3};

                objArr[0] = a1;
                objArr[1] = a1;
                objArr[2] = a1;

                try {
                        for(int i=0; i<objArr.length;i++)
                        {
                                System.out.println(objArr[i].a);
                        }
                } catch (Exception e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
}
Output:
10
10
10
java.lang.NullPointerException
        at
OverloadingnArrays.ArrayNullPointerCls.main(ArrayNullPointerCls.java:26)
```

6. Declaring, Constructing and Initializing an array in a single line

- `int [] intArr = {1,2,3};`
- It is very useful when Array elements are known well in advance.
- Above can be used for Array of objects too

```
Dog puppy = new Dog("Frodo");
Dog[] myDogs = {puppy, new Dog("Clover"), new Dog("Aiko")};
```



Picture demonstrates the result of the following code:

```
Dog puppy = new Dog ("Frodo");
Dog[ ] myDogs = {puppy, new Dog("Clover"), new Dog("Aiko")};
```

Four objects are created:
1 Dog object referenced by puppy and by myDogs[0]
1 Dog[ ] array referenced by myDogs
2 Dog object referenced by myDogs[1] and myDogs[2]

Example :

```
package OverloadingnArrays;

public class TwoDimensionalArray {

	public static void main(String[] args)
	{
		int arr1 [] = {1,3,5};
		int arr2 [] = {2,4,6};
```

```
            int arr [][] = new int [2][2];

            arr[0] = arr1;
            arr[1] = arr2;

            for(int i=0; i<2;i++)
            {
                  for(int j=0;j<2;j++)
                  {
                        System.out.println(arr[i][j]);
                  }
            }
      }
}
Output:
1
3
2
4
```

7. Anonymous Array Initialization :

```
int[] testScores;
testScores = new int[] {4,7,2};
```

Second shortcut for array creation is “anonymous array creation”. Here line one array has been declared and then in second line it has been constructed, initialized and then assigned to the identifier. It is very useful when array is passed as an argument to a method, as shown below.

Ex 1:

```
class ArrMethParam1
{
    void ArrMeth(int [] intArr)
    {
          for(int i = 0; i<intArr.length; i++)
                System.out.println(intArr[i]);
    }
}

public class ArrMethParam {
    public static void main(String[] args)
    {
          ArrMethParam1 amp = new ArrMethParam1();
          amp.ArrMeth(new int []{1,2,3,4});

    }
}
Output:
1
2
3
4
```

- Also remember one cannot specify a size when using anonymous array creation.

```
new Object[3] {null, new Object(), new Object()};
// not legal; size must not be specified
```

Ex.2

```
package OverloadingnArrays;

public class ArrayDeclarationCls
{
	public static void main(String[] args)
	{
		System.out.println("Standard Array initialization");
		int [] arr1 = new int[5];
		for(int i=0; i<arr1.length; i++)
		{
			arr1[i] = i+2;
		}

		for(int j = 0; j<arr1.length; j++)
		{
			System.out.println(arr1[j]);
		}

		System.out.println("Simplest Array Initialization");

		int [] arr2 = {2,4,6,8,10};

		for(int a=0; a<arr2.length; a++)
		{
			System.out.println(arr2[a]);
		}

		System.out.println("Anonymous Array Initialization");

		int [] arr3 = new int[]{1,3,5,7,9};

		for(int b=0; b<arr3.length; b++)
		{
			System.out.println(arr3[b]);
		}
	}
}
Output:
Standard Array initialization
2
3
4
5
6
Simplest Array Initialization
2
4
6
8
10
Anonymous Array Initialization
1
3
```

5
7
9

8. Legal Array Element Assignments :

- Arrays of Primitives: Primitive array can accept any value that could be promoted implicitly to the declared type of the array. For example, an int array can hold any value that could fit into a 32-bit int variable. Thus the following code is legal.

```
int[] weightList = new int[5];
byte b = 4;
char c = 'c';
short s = 7;
weightList[0] = b; // OK, byte is smaller than int
weightList[1] = c; // OK, char is smaller than int
weightList[2] = s; // OK, short is smaller than int
```

- Ex:

```
public class ArrayLegalAssign {

 public static void main(String[] args)
 {
        byte b = 10;
        short s = 20;
        char c = 30;
        int[] intArr = {b,s,c};

        for(int i=0; i<intArr.length; i++)
              System.out.println(intArr[i]);
 }
}
Output:
10
20
30
```

- Arrays of Object references: If declared array type is a class then one can put objects of any subclass of the declared class type. Suppose Ferrari is a subclass of Car then one can put objects of both Ferrari and Car in a array of type Car as follow.

```
class Car {}
class Subaru extends Car {}
class Ferrari extends Car {}
...
Car [] myCars = {new Subaru(), new Car(), new Ferrari()};
```

One should remember that array in array of type Car are nothing but reference variables to the Car Objects. Hence everything that can be assigned to the car reference could be element of array of car objects.

- Ex.

```
class Car
{
  void CarMeth()
```

```
   {
         System.out.println("inside car method");
   }
}

class Ferrari extends Car
{
   void CarMeth()
   {
         System.out.println("inside Ferrari method");
   }
}

public class ArrayLegalAssign {

   public static void main(String[] args)
   {
         byte b = 10;
         short s = 20;
         char c = 30;
         int[] intArr = {b,s,c};

         for(int i=0; i<intArr.length; i++)
               System.out.println(intArr[i]);

         Car car = new Car();
         Ferrari fer = new Ferrari();

         Car[] CarArr = {car,fer};

         for(int j=0; j<CarArr.length; j++)
         CarArr[j].CarMeth();
   }
}
Output:
10
20
30
inside car method
inside Ferrari method
```

- Similarly we can have array of interfaces and we can use all the utilities available for interfaces too. As shown in following example.

```
interface Sporty {
void beSporty();
}
class Ferrari extends Car implements Sporty {
public void beSporty() {
// implement cool sporty method in a Ferrari-specific way
}
}
class RacingFlats extends AthleticShoe implements Sporty {
public void beSporty() {
// implement cool sporty method in a RacingFlat-specific way
}
}
class GolfClub { }
class TestSportyThings {
public static void main (String [] args) {
Sporty[] sportyThings = new Sporty [3];
```

```
sportyThings[0] = new Ferrari(); // OK, Ferrari
// implements Sporty
sportyThings[1] = new RacingFlats(); // OK, RacingFlats
// implements Sporty
sportyThings[2] = new GolfClub(); // NOT ok..
// Not OK; GolfClub does not implement Sporty
// I don't care what anyone says
}
}
```

- The bottom line is this: Any object that passes the IS-A test for the declared array type can be assigned to an element of that array. Also An array declared as an interface type can hold those object references which implements that interface.
- Example :

```
package OverloadingnArrays;

interface ArrayInterface
{
	void meth();
}

class A1 implements ArrayInterface
{
	public void meth()
	{
		System.out.println("Inside A1 meth()");
	}
}

class A2 implements ArrayInterface
{
	public void meth()
	{
		System.out.println("Inside A2 meth()");
	}
}

class A3
{

}

class A4 extends A1
{

}

public class ArrayOfInterfaces
{
	public static void main(String[] args)
	{
		A1 a1 = new A1();
		A2 a2 = new A2();
		A3 a3 = new A3();
		A4 a4 = new A4();
		ArrayInterface ai;
		//ArrayInterface [] ai = {a1, a2, a3, a4};//ERROR: Type
		mismatch: cannot convert from A3 to ArrayInterface
```

```
            //ArrayInterface [] arrIn = {ai, a1, a2, a4};//ERROR: The local
            variable ai may not have been initialized
            ArrayInterface [] arrIn = {a1, a2, a4};
            for(int i=0;i<arrIn.length; i++)
            {
                  ai = arrIn[i];
                  ai.meth();
            }
      }
}
Output:
Inside A1 meth()
Inside A2 meth()
Inside A1 meth()
```

9. Array Reference Assignment :

- One dimentional arrays :
- For Primitives: Array reference means reference to the array object. Don't forget array itself is an object which could hold either primitives or object references of same type. For example, if we define one *int* array then this array reference could only be reassigned to an int array object(of any size) and it not to any other array which is not int.
- Here one should keep in mind that which array element assignment byte could be stored in int array but this rules dont hold true in case of array references. i.e. a int array reference could not be reassign to a byte array.
- Ex:

```
public class ArrayRefAssign {
 public static void main(String args[])
 {
       int[] intArr1 = {1,2,3};
       int[] intArr2 = {4,5,6,7};
       byte[] byteArr = {11,12,13};

       intArr1 = intArr2;
       System.out.println("intArr1 elements are");
       for(int i=0; i<intArr1.length; i++)
       System.out.println(intArr1[i]);

            //intArr1 = byteArr;//Type mismatch : cannot convert from
            byte[] to int[]
 }
 }
 Output:
 intArr1 elements are
 4
 5
 6
 7
```

- For Objects: For reference of arrays which hold objects the rules are not much restrictive. Here the only condition is if it passes IS-A relationship that reference of one object array could be reassign to another.

  Ex :

```
Car[] cars;
Honda[] cuteCars = new Honda[5];
cars = cuteCars; // OK because Honda is a type of Car
Beer[] beers = new Beer [99];
cars = beers; // NOT OK, Beer is not a type of Car
```

- Similarly for interfaces An array declared as an interface type can reference an array of any type that implements the interface. Also remember

  Ex:

```
Foldable[] foldingThings;
Box[] boxThings = new Box[3];
foldingThings = boxThings;
// OK, Box implements Foldable, so Box IS-A Foldable
```

- For Multidimensional Arrays :

  Reference of different dimension of array cannot be assigned to each other. For example, a two-dimensional array of int arrays cannot be assigned to a regular *int* array reference.

```
int[] blots;
int[][] squeegees = new int[3][];
blots = squeegees; // NOT OK, squeegees is a
// two-d array of int arrays
int[] blocks = new int[6];
blots = blocks; // OK, blocks is an int array
```

Chapter 17. Enumerated Types

1) Introduction

➔ An *enum type* is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it. Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week. Because they are constants, the names of an enum type's fields are in uppercase letters.

➔ In the Java programming language, you define an enum type by using the enum keyword. For example, you would specify a days-of-the-week enum type as:

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

➔ Example

```
package enumerationTypes;

//An enumeration of apple varieties.
enum Apple
{
JONATHAN, GOLDENDEL, REDDEL, WINESAP, CORTLAND
}

class EnumDemo
{
  public static void main(String args[])
  {
  Apple ap;
  ap = Apple.REDDEL;

  // Output an enum value.
  System.out.println("Value of ap: " + ap);
  System.out.println();
  ap = Apple.GOLDENDEL;

  // Compare two enum values.
  if(ap == Apple.GOLDENDEL)
  System.out.println("ap contains GoldenDel.\n");

  // Use an enum to control a switch statement.
        switch(ap)
        {
        case JONATHAN:
        System.out.println("Jonathan is red.");
        break;

        case GOLDENDEL:
        System.out.println("Golden Delicious is yellow.");
        break;

        case REDDEL:
        System.out.println("Red Delicious is red.");
```

```
        break;

        case WINESAP:
        System.out.println("Winesap is red.");
        break;

        case CORTLAND:
        System.out.println("Cortland is red.");
        break;
        }
        }
}
Output:
Value of ap: RedDel

ap contains GoldenDel.

Golden Delicious is yellow.
```

2) values() and valuesOf() methods :

➔ All enumerations automatically contains two methods

```
public static enum-type[ ] values( )
public static enum-type valueOf(String str)
```

The **values( )** method returns an array that contains a list of the enumeration constants. The **valueOf( )** method returns the enumeration constant whose value corresponds to the string passed in *str.*

➔ Example :

```
package enumerationTypes;

enum Days {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THRUSDAY, FRIDAY, SATURDAY
}

public class EnumValuesCls {

  public static void main(String[] args)
  {
        Days [] daysArr = Days.values();

        System.out.println("Here is what all Days contains");
        for(Days d : daysArr)
        {
              System.out.println(d);
        }

        System.out.println();
        System.out.println("Using valueOf() method");
        System.out.println(Days.valueOf("FRIDAY"));
        //System.out.println(Days.valueOf("TODAY"));//ERROR:
```

```
  IllegalArgumentException
  }
}
Output:
Here is what all Days contains
SUNDAY
MONDAY
TUESDAY
WEDNESDAY
THRUSDAY
FRIDAY
SATURDAY

Using valueOf() method
FRIDAY
```

➔ In the above code one could directly used converted array from enum as shown below.

```
for(Days d : Days.values())
System.out.println(d);
```

3) <u>Enum as class types</u> :

➔ Enumeration types are class types in Java. They cannot instantiate with 'new' as constructors for enum is only private. Instance of enum is created when Enum constants are first called or referenced in code. Also all enum constants are implicitly static and final and cannot be changed once created. but they have other capabilities as other classes. Because of this enumeration types in Java have much capabilities which other enum types in other languages dont have. For example, you can give them constructors, add instance variables and methods, and even implement interfaces.

➔ It is important to understand that each enumeration constant is an **object** of its enumeration type. Thus, when you define a constructor for an **enum**, the constructor is called when each enumeration constant is created. Also, each enumeration constant has its own copy of any instance variables defined by the enumeration.

➔ <u>Example</u> :

```
package enumerationTypes;

//Use an enum constructor, instance variable, and method.
enum Apple1
{
  JONATHAN(10), GOLDENDEL(9), REDDEL(12), WINESAP(15), CORTLAND(8);
  private int price; // price of each apple
  //Constructor
  Apple1(int p) { price = p; }
  int getPrice() { return price; }
}

public class EnumAsClassType {
```

```
  public static void main(String[] args)
  {
        {
              Apple1 ap;

              // Display price of Winesap.
              System.out.println("Winesap costs "+ Apple1.WINESAP.getPrice() +
              " cents.\n");

              // Display all apples and prices.
              System.out.println("All apple prices:");

              for(Apple1 a : Apple1.values())
              System.out.println(a + " costs " + a.getPrice() + " cents.");
        }
  }
}
Output:
Winesap costs 15 cents.

All apple prices:
JONATHAN costs 10 cents.
GOLDENDEL costs 9 cents.
REDDEL costs 12 cents.
WINESAP costs 15 cents.
CORTLAND costs 8 cents.
```

➔ When the variable **ap** is declared in **main( )**, the constructor for **Apple** is called once for each constant that is specified. Notice how the arguments to the constructor are specified, by putting them inside parentheses after each constant, as shown here:

```
Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);
```

➔ Although the preceding example contains only one constructor, an **enum** can offer two or more overloaded forms, just as can any other class as shown below

```
// Use an enum constructor.
enum Apple1 {
JONATHAN(10), GOLDENDEL(9), REDDEL, WINESAP(15), CORTLAND(8);
private int price; // price of each apple
// Constructor
Apple1(int p) { price = p; }
// Overloaded constructor
Apple1() { price = -1; }
int getPrice() { return price; }
}
```

Notice that in this version, **RedDel** is not given an argument. This means that the default constructor is called, and **RedDel**'s price variable is given the value –1.

➔ Enumerations have only two restriction compare to classes. One they cannot extends other classes and second they cannot be a superclass.

4) java.lang.Enum class

➔ Although you can't inherit a superclass when declaring an **enum**, all enumerations automatically inherit one: **java.lang.Enum**. This class defines several methods which are available for use for all enumerations.

➔ One can get the contants posititon in the enumeration which is called 'ordinal' value using ordinal() method. This and other methods of 'Enum' class like compareTo() and equals() has been demonstrated in the following programme.

```
package enumerationTypes;

enum Dog {LABRADOR, GERMANSHEPHERD, BULLDOG, DOBERMAN, ROTTWEILER}

public class EnumSuperCls {

  public static void main(String[] args)
  {
        Dog dog1, dog2, dog3;

        //Obtain all the values using ordinal
        System.out.println("Ordinal values for all dogs are");
        System.out.println();
        for(Dog d : Dog.values())
              System.out.println(d+"'s ordinal value is : "+d.ordinal());

        dog1 = Dog.BULLDOG;
        dog2 = Dog.LABRADOR;
        dog3 = Dog.BULLDOG;

        //Demonstrate compareTo() and equals()

        System.out.println();

        if(dog1.compareTo(dog2) < 0)//2-0
              System.out.println(dog1+" comes before "+ dog2);

        if(dog1.compareTo(dog2) > 0)//2-0
              System.out.println(dog2+" comes before "+ dog1);

        if(dog1.compareTo(dog3) == 0)//2-2
              System.out.println(dog1+" equals "+ dog3);

        System.out.println();

        if(dog1.equals(dog2))
              System.out.println(dog1+" is "+dog2);
        else
              System.out.println(dog1+" isn't "+dog2);

        if(dog1.equals(dog3))
```

```
                System.out.println(dog1+" is "+dog3);
        else
                System.out.println(dog1+" isn't "+dog3);
    }

}
Output:
Ordinal values for all dogs are

LABRADOR's ordinal value is : 0
GERMANSHEPHERD's ordinal value is : 1
BULLDOG's ordinal value is : 2
DOBERMAN's ordinal value is : 3
ROTTWEILER's ordinal value is : 4

LABRADOR comes before BULLDOG
BULLDOG equals BULLDOG

BULLDOG isn't LABRADOR
BULLDOG is BULLDOG

BULLDOG
```

As we can see above dog1.compareTo(dog2) methods compare the ordinal values of dog1 and dog2 and returns positive if dog1>dog2, 0 if dog1=dog2 and negative if dog1<dog2. The other method equal() compares values contained in e1 and e2 and returns a boolean true or false. Also there is a name() method in Enum class which returns the value contained in enum reference on which it is called.