**Method overloading >**

When a class have same method name with different argument, than it is called method overloading.

**Advantage of method overloading -**

Method overloading enables consistency in the naming of methods which logically perform almost similar tasks and the only difference is in number of arguments. Method overloading enables same method name to be **reused** in program.

Method overloading is done to make program logically more readable and understandable.

**Example -**

|  |
| --- |
| /\*     \* Method to calculate sum of 2 arguments     \*/  **public** **void** **sum(int x, int y)** {            System.*out*.println("sum of 2 arguments = "+ (x+y));     }     /\*     \* Method to calculate sum of 3 arguments     \*/  **public** **void** **sum(int x, int y, int z)** {            System.*out*.println("sum of 3 arguments = "+ (x+y+z));     } |

**sum()** method **logically perform almost similar tasks and the only difference is in number of arguments.** Method overloading enables same method name **sum()** to be **reused** in program.

**10 Features** of Method overloading -

* Call to overloaded method is **bonded** at **compile time**.
* Method overloading concept is also known as **compile time polymorphism** in java.
* Java does **not allow overloading by changing the return type**, though overloaded methods can change the return type.
* Method overloading is **generally done in same class** but **can also be done in SubClass**
* **private methods can be overloaded** in java.
* [**final**](http://www.javamadesoeasy.com/2015/05/final-keyword-in-java-20-salient.html) **methods can be overloaded** in java.
* **Main method can also be overloaded in java**
* **Both** [**Static**](http://www.javamadesoeasy.com/2015/05/static-keyword-in-java-variable-method.html) **and instance method can be overloaded in java.**

Program 1 -  to overload methods by changing number of arguments.

|  |
| --- |
| /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  **public** **class** MyClass {     /\*     \* Method to calculate sum of 2 arguments     \*/  **public** **void** **sum(int x, int y)** {            System.*out*.println("sum of 2 arguments = "+ (x+y));     }     /\*     \* Method to calculate sum of 3 arguments     \*/  **public** **void** **sum(int x, int y, int z)** {            System.*out*.println("sum of 3 arguments = "+ (x+y+z));     }  **public** **static** **void** main(String[] args) {            MyClass obj = **new** MyClass();  **obj.sum(2, 3);** // will call method to calculate sum of 2 arguments  **obj.sum(2, 4, 3);** // will call method to calculate sum of 3 arguments     }  }  /\*OUTPUT  sum of 2 arguments = 5  sum of 3 arguments = 9  \*/ |

Program 2 - to overload methods by keeping same number of arguments **but changing data data type of arguments**

|  |
| --- |
| /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  **public** **class** MyClass {     /\*     \* Method to calculate sum of 2 int type arguments     \*/  **public** **void** **sum(int x, int y)** {            System.*out*.println("sum of 2 int type arguments = "+ (x+y));     }     /\*     \* Method to calculate sum of 2 double type arguments     \*/  **public** **void** **sum(double x, double y)** {            System.*out*.println("sum of 2 double type arguments = "+ (x+y));     }  **public** **static** **void** main(String[] args) {            MyClass obj = **new** MyClass();  **obj.sum(2, 3);** // will call method to calculate sum of 2 int type arguments  **obj.sum(1.2, 2.3);** // will call method to calculate sum of 2 double type arguments     }  }  /\*OUTPUT  sum of 2 int type arguments = 5  sum of 2 double type arguments = 3.5  \*/ |

**Program 3 - Method overloading in SubClass**

|  |
| --- |
| **class** SuperClass{  **public** **void** sum(**int** x, **int** y) {            System.*out*.println("sum of 2 int type arguments = "+ (x+y));     }  }  **class** SubClass **extends** SuperClass{  **public** **void** sum(**double** x, **double** y) {            System.*out*.println("sum of 2 double type arguments = "+ (x+y));     }  }  /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  **public** **class** MyClass {  **public** **static** **void** main(String[] args) {            SubClass obj = **new** SubClass();            obj.sum(2, 1);            obj.sum(2.1, 3.4);     }  }  /\*OUTPUT  sum of 2 int type arguments = 3  sum of 2 double type arguments = 5.5  \*/ |

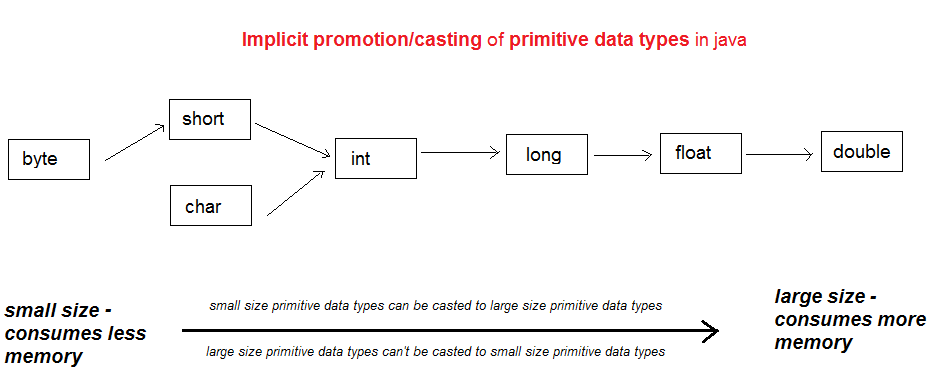
In the above program, method with 2 double type arguments of SubClass **overloaded** method with 2 double type arguments of SuperClass.

**Program 4 - overloading main method in java**

|  |
| --- |
| /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  **public** **class** MyClass {  **public** **static** **void** main(String[] args) {            System.*out*.println("main(String[] args)");  *main*();     }  **public** **static** **void** main(){            System.*out*.println("main()");     }  }  /\*OUTPUT  main(String[] args)  main()  \*/ |

When JVM loads MyClass it finds out main method with signature = [**public** **static** **void** main(String[] args)].

Diagram of **Implicit casting/promotion** of **primitive Data type** in java >



**What is Method overriding in java?**

Method of superclass is **overridden** in subclass **to provide more specific implementation in java**.

**Real time Example of method overriding in java-**

Different animals eat different food, like Lion eat flesh and Goat eat grass. So we can have generic **SuperClass** which tells that **Animal** might eat flesh, grass or may be some other thing.

Now, we can have **SubClasses** like **Lion** which more specifically that **Lion eat flesh**.

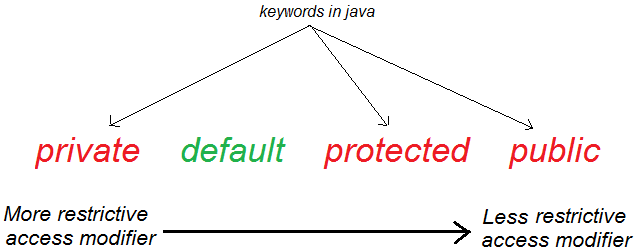
So at runtime, rather than calling food() method of SuperClass, food() method of subclass will be called **and this way we could derive advantage of creating more specific SubClasses and overriding method.** We will elaborate this in **Program 1**.

|  |
| --- |
| /\*  \* superclass - Animal  \*/  **class** Animals {  **void** food() {            System.*out*.println("Animal may eat flesh, grass or ....");     }  }  /\*  \* subclass of Animal - Lion  \*/  **class** Lion **extends** Animals {     @Override  **void** food() {            System.*out*.println("Lion eat - flesh");     }  } |

**Method overriding -** Method of superclass is **overridden** in subclass when overriding method of subclass in java -

1. **Method name -** Overriding method **same name as of superclass method in java,**

1. [**Access modifier**](http://www.javamadesoeasy.com/2015/06/access-modifier-access-specifier-in.html) **-** Overriding method must not have more restrictive access modifier in java.
   * Example 1 - public method cannot be overridden by private method in java.
   * Example 2 - default method can be overridden by default, protected or public method in java.



1. **Return type -** Java **allow method overriding by changing the return type**, but only **Covariant return type** are allowed. (see Program 4)

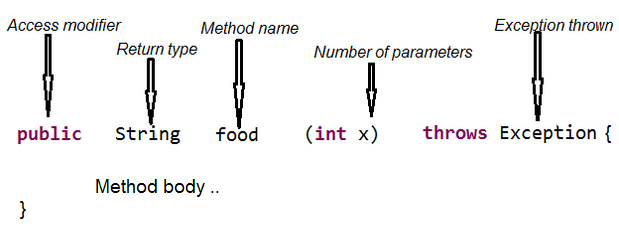
1. **Number of parameters -** Overriding method must have **same number of** [**parameters**](http://www.javamadesoeasy.com/2015/06/difference-between-arguments-and.html) **in java.**

1. [**Exception thrown**](http://www.javamadesoeasy.com/2015/05/throwdeclare-checked-and-unchecked.html) **-** 
   * Overriding method must **not throw new or broader** [**checked exception**](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html) **in java,**
   * though Overriding method may **throw new narrower(subclass) of checked exception or**
   * Overriding method can throw **any runtime exception in java.**

**For more detail on 5th point please Read :** [**Throw/declare checked and unchecked exception while overriding superclass method in java**](http://www.javamadesoeasy.com/2015/05/throwdeclare-checked-and-unchecked.html)

*(Method definition formation) Let’s make above 5 terms easy to remember by diagram /* ***understand method overriding by diagram*** *-*

Method definition is formed by using following 5 terms -



**Note : Return type may be void,** in that case **method doesn’t return anything in java.**

**10 Features** of Method overriding in java -

* Call to overridden method is **bonded** at **runtime** in java**.**
* Method overriding concept is also known as **runtime time polymorphism** in java.
* **Only instance methods can be overridden** in java.
* **private methods can’t be overridden** in java, because private methods are not inherited in subClass.
* [**final**](http://www.javamadesoeasy.com/2015/05/final-keyword-in-java-20-salient.html) **methods can’t be overridden** in java, because final methods are not inherited in subClass.
* [**Static**](http://www.javamadesoeasy.com/2015/05/static-keyword-in-java-variable-method.html) **methods can’t be overridden** in java. ([Please refer this article for detailed analysis and explanation with program](http://www.javamadesoeasy.com/2015/05/why-static-method-cannot-be-overridden.html))
* **Main method also can’t be overridden** in java, because main is static method and static methods can’t be overridden in java (as mentioned in above point)
* It’s important to know that instance variables are never overridden in java.
* Overriding method must not have more restrictive access modifier in java.
* **Method overriding and Exception handling in java** - [**Throw/declare checked and unchecked exception while overriding superclass method in java**](http://www.javamadesoeasy.com/2015/05/throwdeclare-checked-and-unchecked.html)

|  |  |  |
| --- | --- | --- |
|  | ***Method overloading*** | ***Method overriding*** |
| 1 | When a class have same method name with different argument, than it is called method overloading. | Method overriding - Method of superclass is overridden in subclass to provide more specific implementation. |
| 2 | Method is overloaded by - keeping same name of method and only changing number of arguments  Let’s compare with method overriding in java.  Method name - same method name.  Access modifier - Does not matter.           Return type - Does not matter.     Number of parameters in java - Have different number of parameters Exception thrown - Does not matter. | In Method overriding - Method of superclass is overridden in subclass when overriding method of subclass in java -   Method name - Have same name as of superclass method, Access modifier - Must not have more restrictive modifier. Example - public method cannot be overridden by private method.  Return type - Java allow overriding by changing the return type, but only Covariant return type are allowed in java. Number of parameters in java - Have same number of parameters in java. Exception thrown -  Overriding method must not throw new or broader checked exception,  though Overriding method may throw new narrower(subclass) of checked exception or  Overriding method can throw any runtime exception in java.  For more detail on this point please Read : Throw/declare checked and unchecked exception while overriding superclass method in java |
| 3 | Method overloading is generally done in same class but can also be done in SubClass | Method overriding is always done in subClass in java. |
| 4 | Both Static and instance method can be overloaded in java. | Only instance methods can be overridden in java.  Static methods can’t be overridden in java. |
| 5 | Main method can also be overloaded in java | Main method can’t be overridden in java, because main is static method and static methods can’t be overridden in java (as mentioned in above point) |
| 6 | private methods can be overloaded in java. | private methods can’t be overridden in java, because private methods are not inherited in subClass in java. |
| 7 | final methods can be overloaded in java. | final methods can’t be overridden in java, because final methods are not inherited in subClass in java. |
| 8 | Call to overloaded method is bonded at compile time in java. | Call to overridden method is bonded at runtime in java. |
| 9 | Method overloading concept is also known as compile time polymorphism in java. | Method overriding concept is also known as runtime time polymorphism in java. |

Difference between String, StringBuffer and StringBuilder in java - In depth coverage

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Property | **String** | **StringBuffer** | **StringBuilder** |
| 1 | **Storage area** | When String is created **without using new operator**, JVM will create string in [**string pool**](http://www.javamadesoeasy.com/2015/05/string-pool-string-literal-pool-string.html) **area of heap.**  When String is created using **new** operator, it will force JVM to create new string **in heap** (not in string pool).  **Example -->**  **String s1 = "abc";**  **> in string pool area of heap.**  **String s2 = new String("abc");**  **> in heap** | StringBuffer is created in **heap**. | StringBuilder is created in **heap**. |
| **2** | **mutable/** [**Immutable**](http://www.javamadesoeasy.com/2015/05/creating-immutable-class-in-java.html) | [**String is immutable class in java**](http://www.javamadesoeasy.com/2015/05/string-is-immutable-in-java.html), any changes made to Sting class produces new String.    Please see  **example 1a** and **example 1b** below. | StringBuffer is **mutable** class in java, any changes made to StringBuffer class won’t produces new String.  Please see  **example 2** below. | StringBuilder is **mutable** class in java, any changes made to StringBuilder class won’t produces new String.    Please see  **example 3** below. |
| **3** | [**Thread-safe**](http://www.javamadesoeasy.com/2015/03/guidelines-to-thread-safe-code-most.html)**/** [**Synchronized**](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) | **String is immutable that makes it a thread -safe class.** | StringBuffer methods are **synchronized**.  Means no 2 threads on same StringBuffer object **cannot** access methods concurrently. | StringBuilder methods are **not synchronized**.  Means 2 threads on same StringBuilder object **can** access methods concurrently.  **Note** : Methods of StringBuffer and StringBuilder are same, the only difference is of synchronization. |
| **4** | **Performance** | Value of String in String pool is **cached**, hence making it **fast**.  String created with new operator is also **fast** process. | Because of **synchronized methods** its **slow**. | Because of **non synchronized methods** its **fast**. |
| **5** | **Internal working** | Let’s say we have following statements -  *String str = "abc" ;*  *str = str + "def";*  [Internally](http://www.javamadesoeasy.com/2015/05/string-pool-string-literal-pool-string.html) *+* operator uses StringBuffer for concatenating strings.  So,  **String str = new StringBuilder(str).append("def").toString();** | Internally StringBuffer adds new characters to previous StringBuffer. | Internally StringBuilder adds new characters to previous StringBuilder. |
| **6** | **Introduced in** | Java 1  jdk 1.0 | Java 1  jdk 1.0 | Java 5  jdk 1.5 |

**String mutable/** [**Immutable**](http://www.javamadesoeasy.com/2015/05/creating-immutable-class-in-java.html)

**Example 1a -->**

**String str= "ab";**

> No string with “**ab**” is there in string pool, so JVM will create string “**ab**” in **string pool** and **str** will be a reference variable which will refer to it.

**str.concat("cd")**

> **cd** will be concatenated with **ab** and new string  “**abcd**”  will be formed. No string with “**abcd**” is there in pool, so JVM will create string “**abcd**” in **string pool, but** there won’t be any reference variable to “**abcd**”  (we are just using it only in syso statement), meanwhile str will still be pointing to “**ab**”.

**System.*out*.println(str);**

**str** is referring to  “**ab**” and string “**abcd**” will be eligible for [garbage collection](http://www.javamadesoeasy.com/2015/05/finalize-method-in-java-10-salient.html).

**String mutable/** [**Immutable**](http://www.javamadesoeasy.com/2015/05/creating-immutable-class-in-java.html)

**Example 1b -->**

What will happen when below 2 statements will execute >

|  |
| --- |
| **String str= "ab";**  **str = "abcd";** |

**String str= "ab";**

No string with “**ab**” is there in string pool, so JVM will create string “**ab**” in **string pool** and **str** will be a reference variable which will refer to it.

**str = "abcd";**

Now, No string with “**abcd**” is there in string pool, so JVM will create new string “**abcd**” in **string pool** and **str** will be a reference variable which will refer to it.

String **"abcd"** will stay in string pool but reference to it will be lost, and it will be eligible for garbage collection.

**StringBuffer mutable/** [**Immutable**](http://www.javamadesoeasy.com/2015/05/creating-immutable-class-in-java.html)

**Example 2-->**

**StringBuffer sBuffer = new StringBuffer("ab") ;**

>JVM will create stringBuffer object “**ab**” in **heap** and **sBuffer** will be a reference variable which will refer to it.

**sBuffer= sBuffer.append("cd");**

>  “**cd**” will be added to StringBuffer object referred by **sBuffer.** So,   “**abcd**” will be formed.

*(Note: addition was made to previous object, no new object was formed,*

***Behaviour was different as compared to immutable String’s concat function***

*)*

**System.*out*. println (sBuffer);**

**sBuffer** is referring to  “**abcd**” .

**StringBuilder mutable/** [**Immutable**](http://www.javamadesoeasy.com/2015/05/creating-immutable-class-in-java.html)

**Example 3-->**

**StringBuilder sBuilder = new StringBuilder("ab") ;**

>JVM will create stringBuilder object “**ab**” in **heap** and **sBuilder** will be a reference variable which will refer to it.

**sBuilder=sBuilder. append("cd");**

>  “**cd**” will be added to StringBuilder object referred by **sBuilder.** So,   “**abcd**” will be formed.

*(Note: addition was made to previous object, no new object was formed,*

***Behaviour was different as compared to immutable String’s concat function***

*)*

**System.*out*.println(sBuilder);**

**sBuilder** is referring to  “**abcd**” .

**Constructor in java>**

Constructor are used to create object in java.

whenever new keyword is used constructor of class is called to create new object.

1. Constructor have **same name that of class.**

|  |
| --- |
| **public** **class** MyClass {     MyClass(){} //constructor  } |

1. Constructor do **not** have any **return type.**

1. **When no constructor is defined explicitly** in the class, **compiler implicitly provides no-argument constructor** (no-arg constructor) in class.

Let’s say our class is like this -

|  |
| --- |
| **public** **class** MyClass {  } |

compiler will provide default implicit no-arg constructor -

|  |
| --- |
| **public** **class** MyClass {      //default implicit **no-arg constructor** constructor provided by **compiler**.     MyClass(){         super();     }  } |

1. **Constructor chaining** - whenever the object of class is created, **implicitly default no-arg constructor of class and its super class constructor is called.**

Q. But how constructor of superclass is called?

A. Implicitly first statement of constructor is super(), [that means by default first statement of constructor super() is called, super() calls implicit/explicit no-arg constructor of superclass].

Let’s we have superclass and subclass like this -

|  |
| --- |
| **class** SuperClass{  }  **class** SubClass **extends** SuperClass{  } |

compiler will add default implicit no-arg constructor -

|  |
| --- |
| **class** SuperClass{  **SuperClass(){ //no-arg /no argument constructor**  **super();**  **}**  }  **class** SubClass **extends** SuperClass{  **SubClass(){**  **super();**  **}**  } |

Program - Let’s write these implicit statements explicitly to understand flow -

|  |
| --- |
| **class** SuperClass{     SuperClass(){  **super**(); //will call constructor of java.lang.Object (bcz all classes extend Object)            System.*out*.println("constructor of SuperClass");     }  }  **class** SubClass **extends** SuperClass{     SubClass(){  **super**();  //will call constructor of SuperClass            System.*out*.println("constructor of SubClass");     }    }  /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  **public** **class** MyClass{  **public** **static** **void** main(String[] args) {  **new** SubClass(); //will call constructor of SubClass     }  }  /\* OUTPUT  constructor of SuperClass  constructor of SubClass  \*/ |

1. **Instance variables cannot be accessed in constructor until superclass constructors have been called** - Because, as **mentioned in above point**, superclass constructors are called before constructor of class is executed.

1. **If superclass does not contain no-arg constructor** **but contain argument constructor.**

**Than compiler won’t provide no-arg constructor**. To **avoid** compilation error >

* **Solution 1** > subclass must call super(argument) or
* **Solution 2** > declare no-arg constructor explicitly.

Now, we’ll discuss **Solution 1** & **Solution 2** via program.

|  |
| --- |
| **class** SuperClass{     SuperClass(**int** i){ **//Argument constructor**       }    **//Because of argument constructor, compiler won’t provide no-arg constructor**  **}**  **class** SubClass **extends** SuperClass{     SubClass(){       }    } |

Above code will cause compilation error.

In above code - SuperClass does not contain no-arg constructor but contain argument constructor, so compiler won’t provide no-arg constructor. To avoid compilation error >

* **Solution 1** (Elaboration with program) >subclass must call argument constructor of superClass. Ex - super(argument)

|  |
| --- |
| **class** SubClass **extends** SuperClass{  SubClass(){     super(1); **//subclass calling argument constructor of superClass**  }  } |

   Or,

* **Solution 2** (Elaboration with program) > declare no-arg constructor explicitly in SuperClass.

|  |
| --- |
| **class** SuperClass{     SuperClass(**int** i){       }  **//no-arg constructor explicitly in SuperClass**     SuperClass(){       }    } |

1. First line in constructor can either be [super()](http://www.javamadesoeasy.com/2015/06/super-keyword-in-java-invoke.html) or [this()](http://www.javamadesoeasy.com/2015/06/this-keyword-in-java-in-constructor.html). But, super() and this() cannot be used in same constructor. (Please ensure that super() or this() whichever is used must be be first line of constructor, else you will face compilation error)

We read in above points that compiler implicitly adds super() as first line of constructor, but if we add this() explicitly than compiler doesn’t add super()

**Program** to understand usage of this() and super() in constructor and also to show this() and super() cannot be used in same constructor >

|  |
| --- |
| **class** SuperClass{     SuperClass(){           System.*out*.println("no-arg constructor of SuperClass");     }  }  **class** SubClass **extends** SuperClass{     SubClass(){  **this(1);**            System.*out*.println("no-arg constructor of SubClass");     }     SubClass(**int** i){  **super();**            System.*out*.println("int-arg constructor of SubClass");     }    }  /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  **public** **class** MyClass{  **public** **static** **void** main(String[] args) {  **new** SubClass(); //will call constructor of SubClass     }  }  /\* OUTPUT  no-arg constructor of SuperClass  int-arg constructor of SubClass  no-arg constructor of SubClass  \*/ |

1. Constructor can use access modifiers like - **private**, **protected** and **public**. If no access modifier is defined than its **default**.

* private constructor can be used in Singleton classes where object of the class cannot be created outside class.
* class with protected constructor cannot be instantiated in other package. Though constructor can be called through inheritance.

|  |
| --- |
| **public** **class** MyClass {  **private** MyClass(){} //private constructor  }  **public** **class** MyClass {     MyClass(){} //default constructor  }  **public** **class** MyClass {  **protected** MyClass(){} //protected constructor  }  **public** **class** MyClass {  **public** MyClass(){} //public constructor  } |

1. Constructor is not a [keyword in java](http://www.javamadesoeasy.com/2015/05/keywords-in-java-language.html).

1. Interface does not have constructor in java.

1. Constructors are never inherited and hence cannot be overridden.

1. Constructors can be overloaded.

Program to overload constructors >

|  |
| --- |
| **public** **class** MyClass {     MyClass(){ //constructor            System.*out*.println("constructor");     }     MyClass(Integer i){ //overloaded constructor            System.*out*.println("overloaded constructor");     }    **public** **static** **void** main(String[] args) {            MyClass obj1=**new** MyClass();            MyClass obj2=**new** MyClass(1);     }  }  /\* OUTPUT  constructor  overloaded constructor  \*/ |

1. Abstract class also have constructor, and those constructors are called when object of concrete subclass is created, because abstract class cannot be instantiated directly.

|  |
| --- |
| **abstract** **class** AbsClass{     AbsClass(){            System.*out*.println("AbsClass constructor");     }  }  **public** **class** SubClass **extends** AbsClass{     SubClass(){            System.*out*.println("constructor");     }    **public** **static** **void** main(String[] args) {            AbsClass obj1=**new** SubClass();     }  }  /\* OUTPUT  AbsClass constructor  constructor  \*/ |

any attempt to create object of abstract class will generate compilation error.

|  |
| --- |
| AbsClass obj=**new** AbsClass(); //compilation error |

1. If constructor [throws](http://www.javamadesoeasy.com/2015/05/throws-exception-in-java.html) >

* [RuntimeException](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html)/unchecked - Its fine even if not handled.

* CompileTime/checked [Exception](http://www.javamadesoeasy.com/2015/05/exception-handling-exception-hierarchy.html) - It must be caught at time of object creation, or method in which object is created must throw appropriate Exception.

**Example >**

Let’s say constructor throws java.lang.Exception (unchecked)

|  |
| --- |
| **public** **class** MyClass {     //constructor     MyClass() **throws** Exception{     }  } |

Then,

Method in which object is created must throw appropriate Exception.

|  |
| --- |
| /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  **public** **class** MyClass {       //constructor     MyClass() **throws** Exception{     }    **public static void main(String[] args) throws Exception {**  **MyClass obj1=new MyClass();**  **}**    } |

Or,

It must be caught at time of object creation  using [try-catch block](http://www.javamadesoeasy.com/2015/05/try-catch-finally-block-in-java.html).

|  |
| --- |
| /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  **public** **class** MyClass {       //constructor     MyClass() **throws** Exception{     }    **public static void main(String[] args) {**  **try {**  **MyClass obj1=new MyClass();**  **} catch (Exception e) {**  **e.printStackTrace();**  **}**  **}**    } |

Difference between constructors and methods

|  |  |
| --- | --- |
| **constructors** | **methods** |
| Constructors have same name as that of class. | Methods can have same name as that of class, but generally it is bad practice. |
| Constructor do **not** have any **return type.** | Methods have any **return type.**  It method does not return anything its return type is **void**. |
| **Constructor chaining** - whenever the object of class is created, **implicitly default no-arg constructor of class and its super class constructor is called.** | Methods does not have any such chaining. |
| First line in constructor can either be super() or this() | There is no such compulsion with methods. |
| Constructors are never inherited and hence cannot be overridden. | Methods are inherited and hence can be overridden. |

**OOPS :**

***1. Encapsulation***

In short, encapsulation means **data hiding.**

***To achieve encapsulation >***

* Make **fields/memberVariables private (*private*** can be accessed only within the same class, hence we are hiding the fields within the class**)**, and
* **access** those **private fields via public methods**. (***Public*** are accessible from everywhere)

***Advantages of using encapsulation >***

* Encapsulation **prevents other classes to access the class data** (i.e. preventing access to private fields).
* Encapsulation allows to **modify implemented java code without breaking others code** who have implemented the code.

* **Outside users** who are accessing this class **don’t know about the private fields of class**,

***Example*** - field may be Integer or  String type, but user won’t have any such information.

So, **class at any time can change data type of a field and users won’t know about it**, even they need not to. (This point is related to above point)

* Class fields could be made **read-only** or **write-only**.

* ***Encapsulation makes our java code>***
* maintainable,
* extensible and
* flexible.

***Program 1.1 to demonstrate Encapsulation>***

|  |
| --- |
| **class** Employee{  **private** String id; //private field  **public** String getId() { //private field accessed inside public method  **return** id;     }  **public** **void** setId(String id) {  **this**.id = id;     }    }  /\*\* [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  **public** **class** EncapsulationTest {  **public** **static** **void** main(String[] args) {            Employee emp=**new** Employee();            emp.**setId**("1"); //public method can be accessed outside class.            System.*out*.println("emp.getId()  >  "+emp.getId());     }  }  /\* OUTPUT  emp.getId()  >  1  \*/ |

***What would happen without encapsulation>***

No encapsulation means fields won’t be private and could be used outside class.

***Program 1.2 - What would happen without encapsulation >***

|  |
| --- |
| **class** Employee{     String id; //No encapsulation - field isn’t private  }  /\*\* [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  **public** **class** EncapsulationTest {  **public** **static** **void** main(String[] args) {            Employee emp=**new** Employee();            emp.id="1";  //As field isn't private, it could be accessed outside class.       }  } |

***What could be impact of not using encapsulation>***

Let’s say data type of id was changed from String to Integer, in that case compilation error will be generated wherever id has been used, because code was written considering id is String not a Integer.

So, by not using encapsulation we will end up **breaking others code**.

You must be thinking that in below program id has been accessed only at one place, we could make necessary adjustments, but what about id being used at thousands of places in other programs.

***Program 1.3 - impact of not using encapsulation***

**program 1.4 -** Now, lets understand how Encapsulation allows to **modify implemented java code without breaking others code** who have implemented the code via **program**.

If id would have been private, other classes would have been accessing id outside class only via public methods of class. So, in that case if we were to change data type of id from String to Integer than to avoid breaking of others code we could make relevant changes in setter and getter methods.

***In below program >***

As compared to other programs data type of id has been changed from String to Integer,and to avoid breaking of others code we make relevant changes in setter and getter methods. ( Please compare setter and getter methods with above program)

|  |
| --- |
| **class** Employee{     Integer id;  **public String getId() {**  **return String.*valueOf*(id);**  **}**  **public void setId(String id) {**  **this.id = Integer.*parseInt*(id);**  **}**    }  /\*\* [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  **public** **class** EncapsulationTest {  **public** **static** **void** main(String[] args) {            Employee emp=**new** Employee();            emp.setId("1");            System.*out*.println("emp.getId()  >  "+emp.getId());       }  }  /\* OUTPUT  emp.getId()  >  1  \*/ |

***2. Abstraction***

In short, **Abstraction** means **hiding** the **implementation.**

**Abstraction** means representing only the essential things without including background details.

In java abstraction can be achieved by using >

* [**Interfaces**](http://www.javamadesoeasy.com/2015/06/interface-in-java-multiple-inheritance.html) and
* [**Abstract classes**](http://www.javamadesoeasy.com/2015/06/abstract-class-in-java-when-to-use.html)

*Features of Abstract class in java >*

1. Abstract class aren’t **purely abstraction** in java
2. It’s **mandatory** to write abstract keyword to make class abstract.
3. Abstract class **can be abstract** even **without any abstract method**.
4. Abstract class can have priv1ate, [final](http://www.javamadesoeasy.com/2015/05/final-keyword-in-java-20-salient.html), abstract, [static](http://www.javamadesoeasy.com/2015/05/static-keyword-in-java-variable-method.html) and **instance** methods.

**Note** : Method cannot be private and abstract.

Method cannot be final and abstract.

1. Abstract class **have constructors.**
2. Abstract class  **doesn’t allows multiple inheritance** in java.
3. Abstract class  **implements** Interface, **can implement more than one** interface.
4. If any new **instance method** is added in Abstract class then all concrete classes which **extends** that abstract class **need not to provide implementation of newly added instance method**. (see *Program 4* below)

1. If any new **abstract method** is added in Abstract class then all concrete classes which **extends** that abstract class **must provide implementation of newly added abstract method**. (see *Program 5* below)
2. Abstract class can have **synchronized instance methods**, but **abstract methods cannot be synchronized**.

concrete **class which implements abstract class can make abstract methods synchronized**.

1. Even if one abstract method is defined in class than class must be declared as abstract.

*Program 1 -* **Writing your first abstract class**

|  |
| --- |
| /\*  \* abstract class  \*/  **abstract** **class** MyAbstractClass{    **abstract void m();**    }  /\*  \* concrete class  \*/  /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  **class** MyClass **extends MyAbstractClass** {  **@Override**  **public void m(){**  **System.*out*.println("MyClass - m()");**  **}**    **public** **static** **void** main(String[] args) {            MyAbstractClass obj=**new** MyClass();            obj.m();     }  }  /\*OUTPUT  MyClass - m()  \*/ |

In the above program **MyAbstractClass defines abstract method m()**

**and its subClass MyAbstractClass provides implementation of abstract method.**

*Program 2 -* **Writing abstract class with no abstract methods**

|  |
| --- |
| /\*  \* abstract class  \*/  **abstract** **class** MyAbstractClass{    **void m(){**  **System.*out*.println("MyAbstractClass -  m()");**  **}**    }  /\*  \* concrete class  \*/  /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  **class** MyClass **extends MyAbstractClass** {    **public** **static** **void** main(String[] args) {            MyAbstractClass obj=**new** MyClass();            obj.m();     }  }  /\*OUTPUT  MyAbstractClass -  m()  \*/ |

In the above program **MyAbstractClass defines instance method m().**

*When to* ***use abstract class*** *or* ***interface practically -*** *Choosing between* ***interface*** *and* ***abstract class****>*

*When to* ***use abstract class practically*** *-*

Let’s say we have to choose between **class or interface** for **TerrestrialAnimals**, than one thing will be for sure that **habitat of all Terrestrial animals must be land**. That means we can have **same implementation of habitat method for all Terrestrial animals**. And **food of all Terrestrial animals might be different**.

*So, we will* ***create abstract class*** *with -*

***instance method = habitat()*** *[because all Terrestrial animals live on land]*

***abstract method = food()*** *[because food of all Terrestrial animals might be different]*

**instance method habitat()** will be inherited in all subclasses.

*Program 3.1 -* ***use abstract class practically***

|  |
| --- |
| /\*  \* abstract class  \*/  **abstract** **class** TerrestrialAnimals {  **void habitat(){**  **System.*out*.println("Habitat of Terrestrial animal is land");**  **}**    **abstract void food();**    }  /\*  \* concrete class - Lion  \*/  **class** Lion **extends** TerrestrialAnimals {  **@Override**  **void food(){**  **System.*out*.println("Lion eat - flesh");**  **}**  }  /\*  \* concrete class - Lion  \*/  **class** Goat **extends** TerrestrialAnimals {  **@Override**  **void food(){**  **System.*out*.println("Goat eat - grass");**  **}**  }  **public** **class** MyClass{  **public** **static** **void** main(String[] args) {            TerrestrialAnimals lion=**new** Lion();            lion.habitat();            lion.food();            System.*out*.println();              TerrestrialAnimals goat=**new** Goat();            goat.habitat();            goat.food();     }  }  /\*OUTPUT  Habitat of Terrestrial animal is land  Lion eat - flesh  Habitat of Terrestrial animal is land  Goat eat - grass  \*/ |

*When to* ***use interface practically*** *-*

Let’s say we have to choose between **class or interface** for **Animals**, than **habitat of  animals might be land or water**.

And food of all animals might be different.

*So, we will* ***create interface*** *with -*

***abstract method = habitat()*** *[because animals might be living on land or water]*

***abstract method = food()*** *[because food of all animals might be different]*

*Program 3.2 -* ***use interface practically***

|  |
| --- |
| /\*  \* interface  \*/  **interface** Animals {  **abstract void habitat();**    **abstract void food();**    }  /\*  \* concrete class - Lion  \*/  **class** Lion **implements** Animals {  **@Override**  **public void habitat(){**  **System.*out*.println("Habitat of Lion is land");**  **}**    **@Override**  **public void food(){**  **System.*out*.println("Lion eat - flesh");**  **}**  }  /\*  \* concrete class - Lion  \*/  **class** Whale **implements** Animals {  **@Override**  **public void habitat(){**  **System.*out*.println("Habitat of Whale is water");**  **}**    **@Override**  **public void food(){**  **System.*out*.println("Whale eat - aquatic animals");**  **}**  }  /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  **public** **class** MyClass{  **public** **static** **void** main(String[] args) {            Animals lion=**new** Lion();            lion.habitat();            lion.food();              System.*out*.println();              Animals whale=**new** Whale();            whale.habitat();            whale.food();     }  }  /\*OUTPUT  Habitat of Lion is land  Lion eat - flesh  Habitat of Whale is water  Whale eat - aquatic animals  \*/ |

Now, I will like to elaborate 8th point >

*Program 4 -*If any new **instance method** is added in Abstract class then all concrete classes which **extends** that abstract class **need not to provide implementation of newly added instance method**.

|  |
| --- |
| /\*  \* abstract class  \*/  **abstract** **class** MyAbstractClass{    **void m(){**  **System.*out*.println("MyAbstractClass -  m()");**  **}**    }  /\*  \* concrete class  \*/  /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  **class** MyClass **extends MyAbstractClass** {    **public** **static** **void** main(String[] args) {            MyAbstractClass obj=**new** MyClass();            obj.m();     }  }  /\*OUTPUT  MyAbstractClass -  m()  \*/ |

In the above program, MyAbstractClass **defines instance method m()** and concrete classes MyClass which extends that MyAbstractClass **didn’t provide implementation of instance method m()**.

*Program 5 -*If any new **abstract method** is added in Abstract class then all concrete classes which **extends** that abstract class **must provide implementation of newly added abstract method**.

|  |
| --- |
| /\*  \* abstract class  \*/  **abstract** **class** MyAbstractClass{    **abstract void m();**    }  /\*  \* concrete class  \*/  **class** MyClass **extends MyAbstractClass** {  **@Override**  **public void m(){**  **System.*out*.println("MyClass - m()");**  **}**    **public** **static** **void** main(String[] args) {            MyAbstractClass obj=**new** MyClass();            obj.m();     }  }  /\*OUTPUT  MyClass - m()  \*/ |

In the above program, MyAbstractClass **defines abstract method m()** and concrete classes MyClass which extends that MyAbstractClass **provided implementation of abstract method m()**.