OOPS cocepts

1.Data hiding

If we want to restrict the outside person not to access our data directly, We can achieve this by using private modifier. The main advantage of data hiding is we can achieve security.

Ex:

Class Account

{

Private double balance=1000;

Public double getBalance()

{

Return balance;

}

}

Here, from outside the Account class, we can’t access balace variable data directly. We can get that data through getBalance() method only.

2.Abstraction

Hiding internal implementation details is called abstraction. i.e we have to highlight the set of services what we offer without highlighting internal implementation. We can achieve this through interface.

The main advantages of abstraction are

1. Security: as we are not highlighting our internal implementation, there is no chance of knowing our internal logic by outside person.
2. Enhancement will come very easy. without effecting outside person, we can able to change our internal implementation. It improves maintainability.
3. It improves modularity.

3.Encaptulation

If any class follows data hiding and abstraction, such type of class is said to be encapsulated class. i.e encapsulation = data hiding + Abstraction

Hiding data behind method is central concept of encaptulation. See the following example

Example 1:

Class Accout

{

Private double balance;

Public void setBalance(double balance)

{

This.balance=balance;

}

Public double getBalance()

{

Return balance;

}

}

The main advantage of encapsulation is security, but the main limitation is it increases the length of the code and slows down execution.

4.tightly encapsulated class

If every member variable of a class is private, such type class is said to be tightly encapsulated class.

The above example 1 is tightly encapsulated.

The following is not tightly encapsulated because the member variable i is not private.

Class Test

{

Int i=10;

}

Note: if parent is not tightly encapsulated, then no child class is tightly encapsulated. See the following example

Class A

{

Int i=10;

}

Class B extends A

{

Private int j=20;

}

Class C extends B

{

Private int k=30;

}

5.Is-a relation

1. It is also known as inheritance.
2. By using extends keyword, we can implement it.
3. Reuseability is the biggest advantage of inheritance.
4. Whatever the parent has by default available to child, but whatever child has by default not available to the parent.
5. Parent class reference can be used to hold child class object, but child class reference can’t hold parent class object.
6. Parent class reference can hold the child class object, but we are not allowed to call child class’s specific method.

Ex:

Class A

{

Int i=10;

}

Class B extends A

{

Int j=20;

Public static void main(String args[])

{

A a1=new B(); // valid as per rule no 5

B b1=new A(); // invalid as per rule no 5

B b1=new B();

System.out.println(b1.i); // valid as per rule no 4

System.out.println(a1.i); // is correct.

System.out.println(a1.j); // is not correct as per rule no 6.

}

}

7. java doesn’t support cyclic inheritance, otherwise compile time error.

Cyclic inheritance example is the following.

Example:

Class A extends B

{

}

Class B extends A

{

}

6.has-a relation:

1. It is also known as composition or aggregation or deligation. There is no specific keyword to implement. But mostly we can achieve this by using new keyword.
2. This relationship increases dependency between the components and reduces the maintainability. See the following example.

Class A

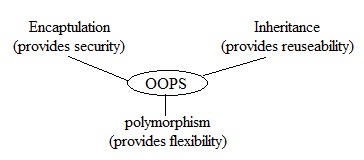
{  
 B b1=new B(); // observe this statement. Here it has complete B class object.

}

Class B

{

}



7.method signature:

1. Method signature consists of method name followed by argument list,and the order of arguments is also important.
2. Return type is not part of signature in java.
3. For example

Public int m1(int i , float f ) { } //Here, m1(int , float) is the signature.

1. Method signatures are used by compiler while resolving method calls.
2. Two methods having the same signature are not allowed in any java class, violation leads to compile time error.

For example

Clas Test

{

Public void m1(int x) // here the signature is m1(int)

{   
 }

Public void m1(int y) // here the signature is m1(int)

{

}

}

Compile time error, saying m1(int) is already defined.

8.Overloading:

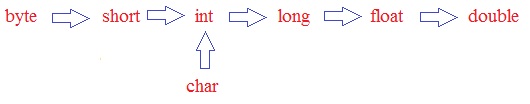
1. it is nothing but compile time polymorphism or static polymorphism or early binding.
2. In C language, two methods having the same name are not allowed. If we are changing the argument, then compulsory we should go for changing method name with different name. due to that reason the complexity of C-programming is increase.
3. but in java, we can use the same method name with different arguments. This kind of method is called as overloaded method.

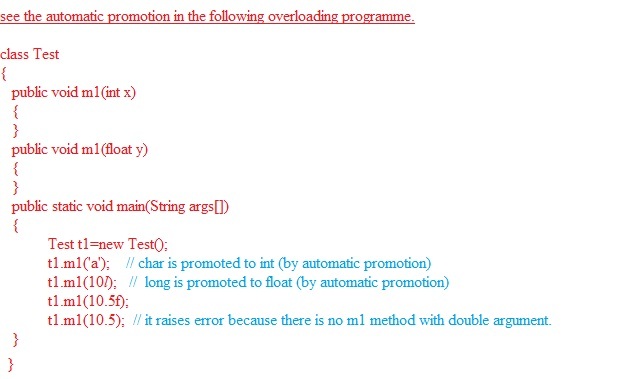
Ex: fun(), fun(int), fun(int, int), fun(float), fun(int, float)….

1. in overloading, The method resolution always takes care by compiler, based on reference type and arguments.hence overloading is also considered as compile time polymorphism, static polymorphism, early binding.
2. While overloading method resolution, if the compiler unable to find the method with required argument, it won’t raise compile time error immediately.

First compiler promotes argument to next level and checks whether the matched method is available or not. If it is not available, then compiler once again promotes the argument to next level and checks for the matched method. This process will be continued until all possible promotions, still if the compiler is unable to find the matched method, then only it will raise compile time error.

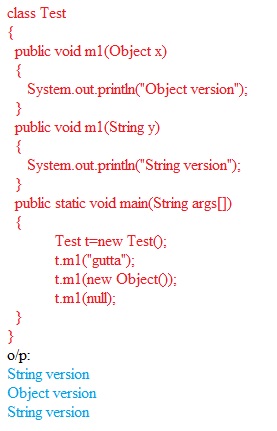
The following is the list of all possible automatic promotions.

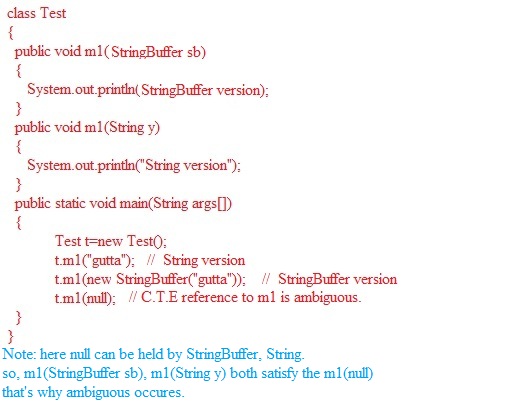




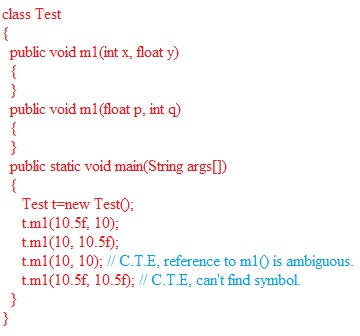
1. In overloading method resolution, child will always get high priority.

See the following example.

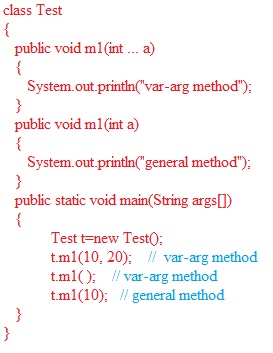




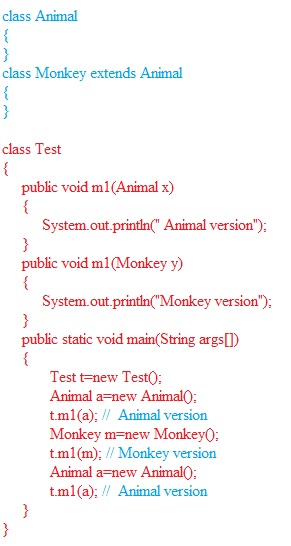
1. In overloading method resolution, if arguments are at same level, then we will get compile time error.



1. In general, var-arg method will always get least priority. i.e if no other method is matched, then only var-arg method will be considered.

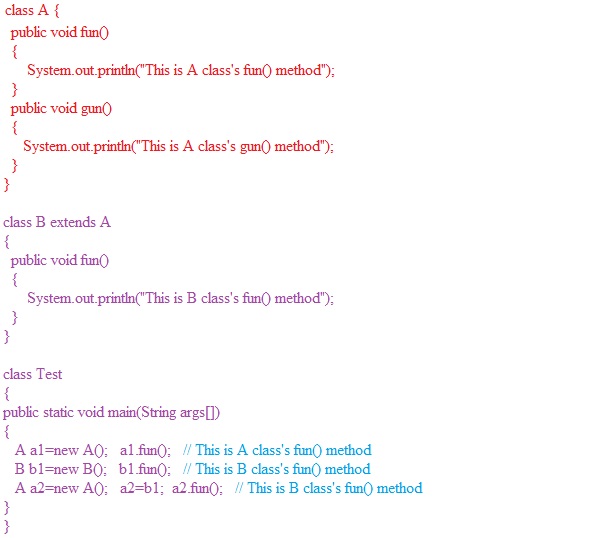


1. While overloading metod, compiler always considers only references, but not runtime object.



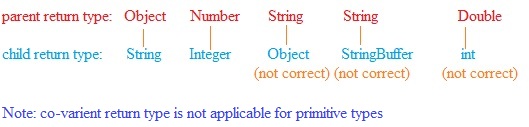
9.Overriding:

1. It is nothing but dynamic polymorphism, runtime polymorphism, late binding.
2. Whatever the parent has by default available to the child. If the child is not satisfied with parent class implementation, then child is allowed to override parent class implementation in its own way. This is concept is nothing but overriding.
3. If parent class method which is overridden is called overridden method and child class method is called as overriding method.
4. Overriding method resolution will always take care by jvm based on runtime object, hence overriding is also considered as runtime polymorphism or dynamic polymorphism or late binding.

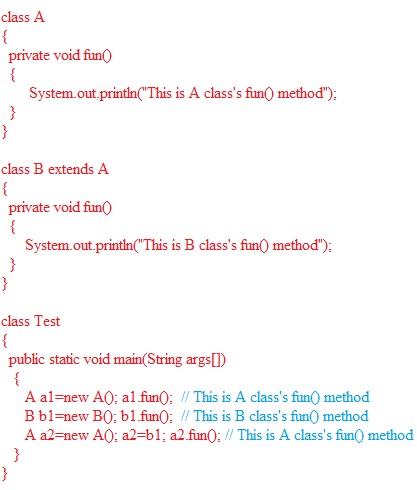


1. Rules for overriding
2. Method names and argument must be the same including order i.e signature of the methods must be the same.
3. While overriding return types must be the same but this rule is applicable only upto 1.4 version. But from 1.5 version onwards co-varient return types also allowed, according to this, child method return type need not to be the same as parent method return type. It’s child class is also allowed. For clarity see the following

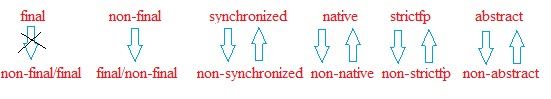
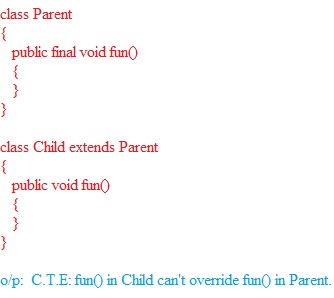
Example:



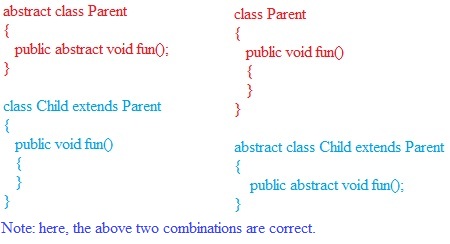
1. Private methods are not visible in child classes hence overriding concept is not applicable for private methods. Based on requirement we can take exactly the same method in child class, but it’s not overriding.



1. We can’t override final methods of parent class. Violation leads compile time error.

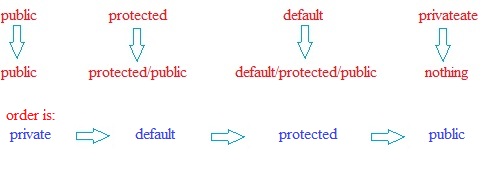


Observe the following example



From the above example, We can override parent class abstract method in child class to provide implementation. And we can override parent class’s concrete method with child class’s abstract method.

(v) While overriding, weakering of access modifier is not allowed.



For example

Class Parent

{

Public void fun()

{

}

}

Class Child extends Parent

{

Protected void fun() // due to the protected modifier, it raises C.T.E

{

}

}

(vi)checked versus unchecked exception

The exceptions which are checked by the compiler for smooth execution of program at runtime are called checked exception. Ex: IOException, InterruptException

The exceptions which are not checked by the compiler are called unchecked exception.

Ex: ArithmeticException, NullPointerException

Runtime Exception and its child classes, error and its child classes are unchecked and all the remaining are considered as checked exception.

Whether the exception is checked or unchecked, it should always occurs at runtime, there is no chance of raising an exception at compile time.

Example:

Class Parent

{

Public void fun() throws IOException

{

}

}

Class Child extends Parent

{

Public void fun() throws IOException, ArithmeticException

{

}

}

The above code is valid, because ArithmeticException is unchecked exception.

Note:

While overriding, the size of the checked exception and level of checked exception can be decreased, but we are not allowed to increase.

Example:

Class Parent

{

Public void fun() throws IOException

{

}

}

Class Child extends Parent

{

Public void fun() throws IOException, InterruptException

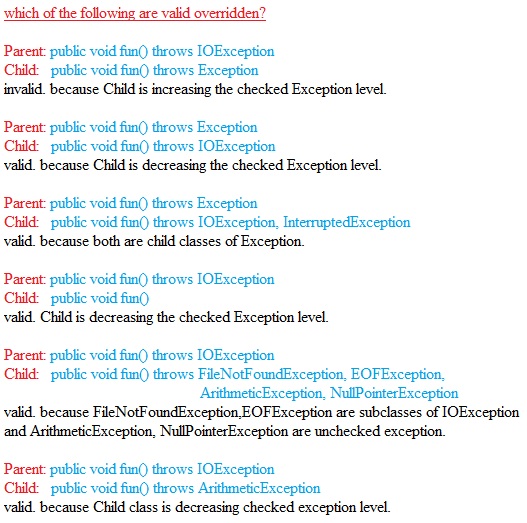
{

}

}

The above code is invalid. It raises error like the following

Fun() in Child can’t override fun() in Parent; overridden method doesn’t throw InterruptedException.



(vii) overridding withrespect to static method:

Parent: public static void fun()

Child: public void fun()

Invalid, C.T.E: fun() in Child can’t override fun() in Parent, overridden method is static.

We can’t override static method with non static and vice versa. i.e

Parent: public static void fun()

Child: public void fun()

Invalid, C.T.E saying overridden method is static.

Parent: public void fun()

Child: public static void fun()

Invalid, C.T.E saying overriding method is static.

If both parent and child class methods are static, then we won’t get any compile time error, but it is not overridden and this new concept is called method hiding.

Parent: public static void fun()

Child: public static void fun()

Valid, Parent class’s method won’t be overridden.

Method hiding:

It is exactly same as the overriding, except the following difference.

That is both parent and child class should be static. Method resolution will always take care by compiler based on reference type(i.e if you call with child reference, then child class’s method will be executed. if you call with parent reference, then parent class’s method will be executed)

All the rules of method hiding and overriding are exactly the same but the following differences

|  |  |
| --- | --- |
| Method overriding | Method hiding |
| 1.both parent and child class’s method should be not static. | 1.both parent and child class’s method should be static. |
| 2.method resolution always takes care by jvm based on runtime object. | 2.method resolution always takes care by compiler based on reference type. |
| 3.it is considered as runtime polymorphism or dynamic polymorphism or late binding. | 3. it is considered as static polymorphism or compile time polymorphism or early bining. |

Note:

Overriding concept is not applicable for variables. Variable resolution will take care by compiler based on reference type.

Difference between overloading and overriding.

|  |  |  |
| --- | --- | --- |
| properties | overloading | Overriding |
| Method name must be the same | Must be the same | Must be the same |
| Arguments | Should be different at least order. | Should be the same including order. |
| Signature | Must be different | Must be the same |
| Return type | No ristriction | Must be same upto 1.4, co-varient return types allowed from 1.5 |
| Access modifier | No restriction | Weakering access modifiers not allowed. |
| Throws class | No restriction | The size and level of checked exception should be decrease. No restrictions on unchecked exceptions. |
| Private, final, static methods | Can be overloaded | Can’t be overridden |
| Method resoultion | Always takes care by compiler based on reference type | Always takes care by jvm based on runtime object |
| Also known as | Static polymorphism, compile time polymorphism, early binding | Dynamic polymorphism, runtime polymorphism, late binding |

Q: Parent: public void fun(int i) throws IOException, for this which of the child class’s methods are valid as per the above overloading, overriding rules?

1. Private void fun() throws IOException
2. Private void fun(int i) throws Exception
3. Public void fun()
4. Public static void fun(long l)
5. Public static final abstract void fun(long l)
6. Public static void fun(int i)
7. Public void fun(int i) throws IOException, ArithmeticException

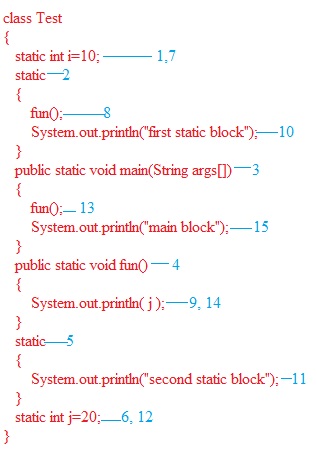
Answer: as per overloading rules, 1,3,4 are correct.

As per overriding rules, 7 is correct. Remaining all are invalid for polymorphism.

Q: tell me the static control flow?

1. Identification of static members from top to bottom.
2. Execution of static variables assignment and static blocks from top to bottom
3. Execution of main method

See the following example to understand the static control flow.

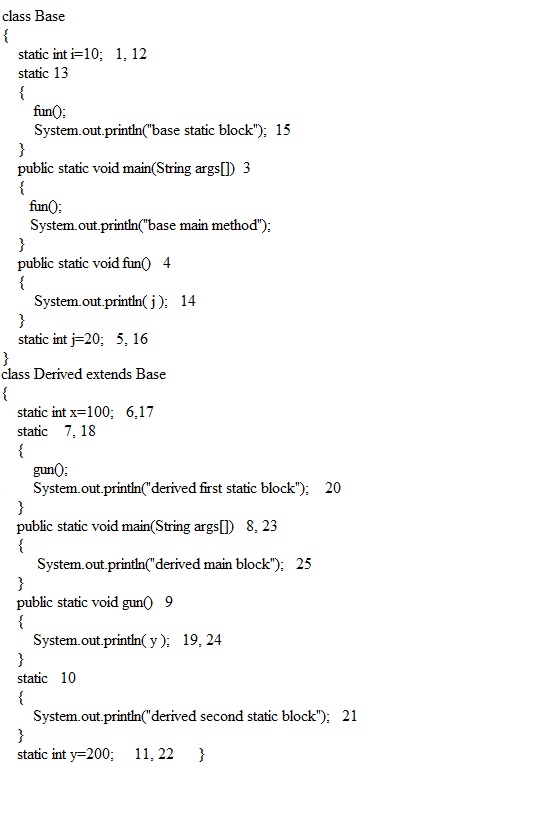


Q: static control flow in parent and child class relationship?

1. Identification of static members from parent to child
2. Execution of static variable assignments and static blocks from parent to child
3. Execution of the main method of child class only.

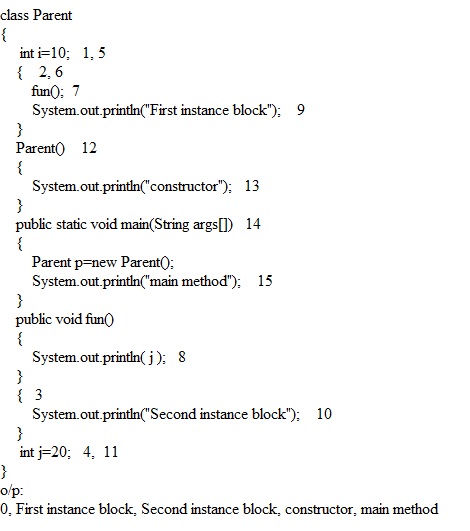
Note: whenever we are loading child class, automatically parent class will be loaded. But whenever we are loading parent class, child class is not required to load.

1. See the following example to understand static control flow in parent & child class relation



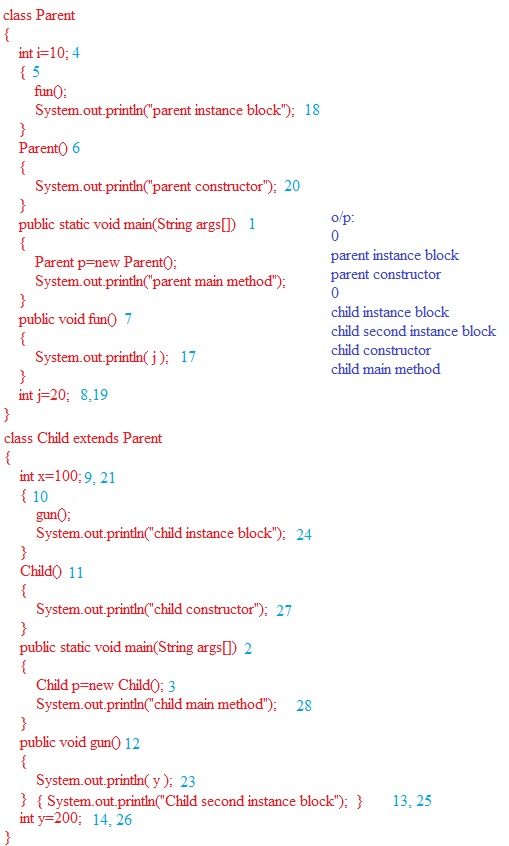
Q: tell me the instance control flow?

1. Identification of instance members from top to bottom
2. Execution of instance variable assignment and instance blocks from top to bottom.
3. Execution of constructor
4. See the following example



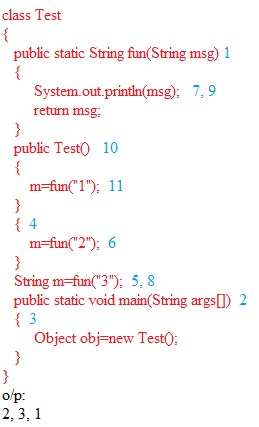
Q:Instance control flow in parent & child class relation

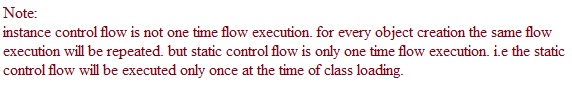
1. Identification of instance members from parent to child
2. Execution of instance variable assignment and instance blocks only in parent class.
3. Execution of parent class constructor
4. Execution of instance variable assignment and instance blocks in child class.
5. Execution of child class constructor
6. See the following example to understand the flow



Q: if class is combination of static and instance variables, blocks then

1. Identification of static members from top to bottom.
2. Execution of static variables assignment and static block from top to bottom.
3. Execution of main()
4. Identification of instance members from top to bottom
5. Execution of instance variables assignment and instace block from top to bottom.
6. Execution of constructor

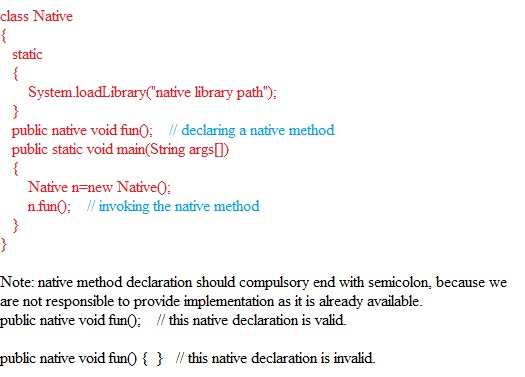




Q: tell me about native keyword

1. Native is the modifier applicable only for method and we can’t apply for variables and classes.
2. The method which are implemented in non java are called native methods or foreign methods.
3. The main objectives of using native keyword are to improve performance of the system, to use already existing legacy code (C, C++)

Example:



1. For native method, implementation is already available, but for abstaract method, implementation should not be available. Hence abstract native combination is illegal for methods.
2. For native methods inheritance, overloading, overriding concepts are applicable.
3. Native methods can’t be declared as strictfp.
4. The usage of native keyword breaks platform independent nature of java.

Q: tell me about volatile keyword?

1. Volatile is the modifier applicable only for variables, but not for methods and classes.
2. If the value of a variable is kept on changing, such kind of variables we have to declare with volatile modifier.
3. If the variable declared as volatile, then for every thread a separate local copy will be created. All intermediate modification performed by that thread will take place in local copy instead of master copy. Before terminating that thread, the final value will be updated in the master copy.
4. The main advantage of volatile keyword is we can prevent data incosistancy problem. But creating and maintain a separate copy for every thread increases complexity of the programming and effects on performance of the system. Hence volatile keyword is not recommended to use.
5. Volatile variable means its value keep in changing where as final variable means its value should not be change. Hence final volatile combination is illegal for variables.

Q: summary on access modifiers

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Outer class | Inner class | methods | variables | blocks | Constructors |
| Private | No | yes | yes | yes | no | Yes |
| Default | yes | yes | yes | yes | no | Yes |
| protected | No | yes | yes | yes | no | Yes |
| Public | yes | yes | yes | yes | No | Yes |
| Final | yes | yes | yes | yes | no | No |
| Static | no | yes | yes | yes | yes | No |
| Abstract | yes | yes | yes | no | no | No |
| Synchronized | no | no | yes | no | yes | No |
| Native | no | no | yes | no | no | No |
| Strictfp | yes | yes | yes | no | no | No |
| Transient | no | no | no | yes | no | No |
| Volatile | no | no | no | yes | no | No |

10.Interface:

1. from the client point of view, an interface define as the set of services what he is expecting.
2. From the service provider point of view, an interface define the set of services what he is offering. Hence interface acts as contract between client and service provider.
3. Every method present inside interface is abstract and we should not provide any implementation inside interface. Hence an interface is also considered as 100% pure abstract class.
4. The main advantages of interface are
5. We can achieve security as we aren’t highlighting our internal implementation
6. With out effecting outside world we can perform any type of modification to our internal system.
7. Enhancement will become very easy.
8. Two different system can able to communicate through interfaces. Eg: A java system can communicate with main frame system.
9. Interface declaration and implementation: we can declare an interface, by using interface keyword, a class can implements an interface by implents keyword.
10. Whenever a class implements an interface, compulsory we should provide implementation for every interface method. Otherwise we have to declare that class as abstract. Violation leads to compile time error.
11. Whenever we are implementing an interface method compulsory, we should declare that method as public, otherwise we will get compile time error.
12. The only applicable modifiers for the interface are public, default, abstract, strictfp. If we use any other modifier, we will get compiletime error.
13. Extends vs implements: a class can extend only one class at a time but an interface can extend any no of interfaces at a time. And a class can implement any no of interfaces at a time.

Q: which of the following are true?

1. A class can extend any no of classes at a time.
2. An interface can extend only one interface at a time.
3. An interface can implement any no of interfaces at a time
4. A class can extends any no of interfaces
5. A class can implement any no of interfaces and can extends any no of classes simultaneously

Answer: All the above are wrong.

Q: consider the expression X implements Y,Z for which of the following possibilities of X,Y,Z is true.

1. X,Y,Z should be classes.
2. X,Y,Z should be interfaces.
3. X should be the class, and Y,Z should be interfaces.
4. None of the above

Answer: 3 is correct.

Note: X extends Y,Z. this is true, when X,Y,Z are interfaces.

Note: X extends Y. this is true, when X,Y are either classes or interfaces.

Interface methods:

Every interface method is bydefault public and abstract whether we are declaring or not. Hence the following method declarations are equal inside interface.

1. Void fun();
2. Public void fun();
3. Public abstract void fun();
4. Abstract void fun();

Q:Which of the following method declarations are legal inside an interface?

1. Public void fun(){ }
2. Private void fun();
3. Protected void fun();
4. Public static void fun();
5. Public abstract void fun();

Answer: only 5 is correct.

Interface variables:

An interface can also contain variables.

Every interface variable is by default public, static and final whether we are declaring or not. Hence the following variable declarations are equal inside an interface.

1. Public int i=10;
2. Int i=10;
3. Public static int i=10;
4. Public static final int i=10;

Note: for the interface variable, we should compulsory perform initialization at the time of declaration otherwise we will get compiletime error.

Interface X {

Int i; // it raises C.T.E illegal start of expression.

}

As the interface variables are by default public, static, final. We are not allowed to declare with the following modifiers.

Private, protected, volatile, transient

Q:which of the following are correct?

1. Int i;
2. Int i=10;
3. Public transient int i=10;
4. Public static final int i=10;
5. Volatile int i=10;

Answer: 2, 4 are correct.

Interface method naming conflicts cases:

Case (i):

If two interfaces contain a method with same signature and same return type then in the implementation class, we have to provide implementation for only one method.

Example:

Interface gface

{

Public void fun();

}

Interface mface

{

Public void fun();

}

Class Test implements gface, mface

{

Public void fun()

{

}

}

Case (ii):

If two interfaces contain a method with same name but different arguments then in the implementation class, we have to provide implementation for the two methods. These methods act as overloaded methods.

Example:

Interface gface

{

Public void fun();

}

Interface mface

{

Public void fun( int x);

}

Class Test implements gface, mface

{

Public void fun()

{

}

Public void fun(int x)

{

}

}

Case (iii):

If two interfaces contain a method with same signature but different return types, then we can’t provide implementation for both simultaneously.

Interface gface

{

Public void fun();

}

Interface rface

{

Public int fun();

}

There is no way to provide implementation for both interfaces simultaneously.

Q: Is a java class can implement any no of interfaces simultaneously?

Yes, except two interfaces contain a method with same signature but different return types.

Variable method naming conflicts

if two interfaces contain a variable with the same name, then there may be a chance of getting variable maning conflicts. But we can resolve interface variable conflicts by using interface name.

Interface gface

{

Int i=111;

}

Interface rface

{

Int i=222;

}

Class Test implements gface, rface

{

System.out.println(i); // raises C.T.E, saying reference to i is ambiguous.

System.out.println(gface.i);

System.out.println(rface.i);

}

Marker interface:

by implementing an interface, if our object gets some ability, such type of interfaces are called as marker interface. It is also called as ability interface, tag interface.

Ex: Serializable, Cloneable, RandomAccess, Comparable, SingleThreadModel,….

Note: marker interface may have methods or may not have methods.

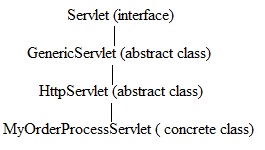
Q: when should we go for interface, abstract class and concrete classes?

If we don’t know about implementation and just we have specification then we should go for interface.

If we are talking about implementation, but not complete then we should go for abstract class.

If we are talking about implementation completely and ready to provide service then we should go for concrete class.

See the following example



Q: What is the difference between interface and abstract class?

|  |  |
| --- | --- |
| interface | Abstract class |
| If we never talks about implementation and just we have requirement specification, then we should go for interface. | If we are talking about implementation, but not complete then we should go for abstract class. |
| An interface should contain only abstract methods. Hence interface considered as 100% pure abstract class. | an abstract class can contain concrete methods also. |
| Every varialble present inside an interface is public static final. | Variable inside an abstract class can be non static, non final also. |
| Interface can’t have constructors. | Abstract class can have constructors. |
| Interface variables can’t be private, protected, volatile and transient. | Abstract class variables can be private, protected, volatile and transient. |
| Interface methods never be private, protected, final, synchronized, native, strictfp, static. | Abstract class methods can be private, protected, final, synchronized, native, strictfp, static. |

Adoptor classes

the class which is implementing an interface with dummy or empty implementation such type of classes are called as adaptor class. Instead of implementing interface directly, we have to extends adopter classes.

In servlet specification, GenericServlet acts as adaptor class.

11.Constructors

1. Object creation is not enough, compulsory we should perform initialization, then only the object is a position to provide service.
2. Whenever we are creating an object, some piece of code will be executed internally to perform initialization, that piece of code is nothing but constructor.
3. The main objective(duty) of constructor is to perform initialization.
4. For example,

Class Student

{

String sname;

Int rollno;

Student(String sname, int rollno)

{

This.name=name;

This.rollno=rollno;

}

Public static void main(String args[])

{

Student s1=new Student(“gutta”, 101);

Student s2=new Student(“suresh”, 102);

}

}

1. Rules for writing constructor
2. The name of the constructor and name of the class must be the same.
3. The only applicable modifiers for the constructor are private, public, protected, default. If we are using any other modifier, we will get compilte time error.

Class Test

{

Final Test()

{

} // C.T.E saying, modifier final not allowed here.

}

1. Return types are not allowed for constructors, even void also. If we are defining return type then the compiler treats it as a method and we won’t get any compile time error or runtime error.

Example:

Class Test

{

Void Test() // compiler treats this Test() as a method, but not like constructor.

{

}

}

1. In general, private constructors are useful to define singleton classes. For any class, if we are allowed to create only one object. Such type of classes are called as singleton classes. We can create our own singleton class also.

Example for singleton class

Class Test implements Cloneable

{

Private static Test t=null;

Private Test() {}

Public static Test getInstance()

{

If (t==null)

. t=new Test();

Else

Return t;

}

}

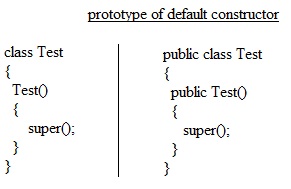
What is the advantage of singleton class?

Instead of having multiple objects, having only one object improves memory utilization and performance of the system.

Default constructor:

if we are not writing any constructor, then compiler will always generates a default constructor.

If we are writing any constructor, then compiler won’t generate any default constructor i.e every class should contain either compiler generated constructor or programmer written constructor, but not both simultaneously.



1. Default constructor is always no argument constructor.
2. It contains only one line super()
3. It is no argument call to the super class constructor.
4. The modifier of the default constructor is same as class modifier. This rule is applicable only for public and default. See the above example for clarity.

|  |  |
| --- | --- |
| Programmer’s code | Compiler generated code |
| Class Test  {  } | Class Test  {  Test()  {  Super();  }  } |
| Public class Test  {  } | Public class Test  {  Public Test()  {  Super();  }  } |
| Class Test  {  Void Test()  {  }  } | Class Test  {  Test()  {  Super();  }  Void Test()  {  }  } |
| Class Test  {  Test(int i)  {  }  } | Class Test  {  Test(int i)  {  Super();  }  } |
| Class Test  {  Test(int i)  {  Super();  }  } | Class Test  {  Test(int i)  {  Super();  }  } |
| Class Test  {  Test(int i)  {  this();  }  Test()  {  }  } | Class Test  {  Test(int i)  {  this();  }  Test()  {  Super();  }  } |

Super() & this()

the first line inside any constructor either super or this. If we are not taking any thing then the compiler will always play super().

Case (i):

Class Test

{

Test()

{

System.out.println(“constructor”);

}

Super(); // C.T.E saying call to super must be first statement in constructor.

}

Case (ii):

Class Test

{

Test()

{

Super();

This(10); // C.T.E saying call to this() must be the first statement.

System.out.println(“constructor”);

}

Test(int i)

{

}

}

Case (iii):

class Test

{

Void fun()

{

Super(); // C.T.E saying super() must be the first statement in constructor.

}

}

Case (iv):

Class Test

{

Void fun()

{

System.out.println(this.hashCode()); // valid.because super,this keywords are separate compare to super(), this().

}

}

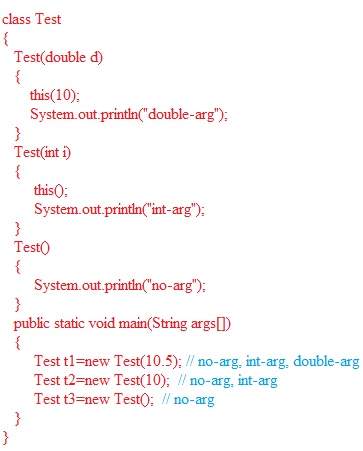
Note:

Super(), this() we should use only in the constructor as first statement. But not both simultaneously.

Super, this keywords are separate compare to super(), this().

Overloaded constructor:

a class contain multiple constructors and these constructors are considered as overloaded constructors. For example



Parent class constructors won’t available to child i.e inheritance concept is not applicable to the constructor. Hence overriding concept is also not applicable.

Every class in java, including abstract class contain constructor concept but interface doesn’t contain any constructor.

|  |  |  |
| --- | --- | --- |
| Valid | valid | Invalid |
| Class Test  {  Test()  {  }  } | Abstract class Test  {  Test()  {  }  } | interface Test  {  Test()  {  }  } |

Case (i):

Recursive method call is always RuntimeException saying StackoverFlow error. Where as Recursive constructor invocation is a compile time error. See the following example

Class Test

{

Test()

{

This(10);

}

Test(int i)

{

This();

}

} // C.T.E saying recursive constructor chaining.

Case (ii):

Class Parent

{

Parent()

{

}

}

Class Child extends Parent

{

} // this code is valid.

Note: whenever we are writing some argument constructor, it is always a good programming practice to write no argument constructor also.

Case (iii):

Class Parent

{

Parent(int i)

{

Super();

}

}

Class Child extends Parent

{

/\* here compiler generates defaultly the following code

Child()

{

Super();

}\*/

} // C.T.E saying no default constructor in Parent class.

Note: if the parent class contains some constructors, then while writing child classes we should take special care with respect to constructor.

Case (iv):

if parent class constructor throws some checked exception compulsory child class constructor should through the same checked exception or its parent otherwise we will get compile time error. But this is rule is not applicable for unchecked exceptions.

|  |  |
| --- | --- |
| No compile time errors in the following code. Because the throughing exception is unchecked. | C.T.E saying un reported exception in default constructor |
| Class Parent{  Parent() throws ArithmeticException  {  }}  Class Child extends Parent  {  } | Class Parent{  Parent() throws Exception  {  }}  Class Child extends Parent  {  } |

All the following statements are false

1.compiler will always generate default costructctor.

2.the first line inside any constructor should be this() always.

3.the first line inside any constructor should be either super() or this(). If we are not writing anything then compiler places this().

4.for constructors both overloading and overriding is applicable

5.constructors can be overridden.

6.recursive constructor call is always a runtime exception.

7.if the parent class constructor throws some checked exception, then child class constructor is not required to through that checked exception.

8.if the parent class constructor throws some unchecked exception, then child class constructor should through that unchecked exception.

Coupling:

the degree of dependency between the components is called coupling.

Example:

|  |  |
| --- | --- |
| Class A  {  Int i=B.j;  } | Class B  {  Static int i=C.fun();  } |
| Class C  {  Public static int fun()  {  Return D.k;  }  } | Class D  {  Static int k=10;  } |

The above four components are said to be tightly coupled with each other and this is not a good programming practice. This approach has several serious disadvantages.

It reduces maintainability

Enhancement will become very complex, because without effecting remaining components, we can’t modify any component.

It doesn’t promote reuseability.

It is always a good programming practice to maintain a loosely coupling between the components.

Type casting:

1. parent class reference can be used to hold child class object, but by using that reference we are not allowed to call child class specific methods.
2. Interface reference can be used to hold its implemented class object but by using that reference we are allowed to call the methods present in that interface and we can’t call implemented class specific methods.
3. Signature of the type casting:

A b=( c )d;

A is class/ interface, b is name of reference variable, c is class/interface, d is name of the reference variable/object.

Compile time checking 1:

The type of d and c must have some relationship either parent to child or child to parent or the same type. Otherwise we will get compiletime error saying inconvertible types.

Example: the following is valid

Object o=new String(“gutta”);

StringBuffer sb=(StringBuffer)o;

Example: the following is invalid because there is no relation between String and StringBuffer.

String s=new String(“gutta”);

StringBuffer sb=(StringBuffer)s;

Compiletime checking 2:

c must be same or derived type of A or child type of A.otherwise we will get compile time error saying incompatable types found.

Runtime checking:

The type of d and c must have some relationship either parent to child or child to parent or the same type. Otherwise we will get compiletime error saying inconvertible types.

Example: the following won’t raise any compile time error but it raises runtime error.

Object o=new String(“gutta”);

StringBuffer sb=(StringBuffer)o;

Note: whenever we are trying to perform type casting just we are changing the type of reference variable and we are not coverting internal object type.

It the above example, the internal object in o reference variable is String. That’s why we couldn’t change it into StringBuffer.

Note: variable resolution always takes care by compiler depends on the reference type, but not runtime object. For clarity on this, observe the above example.

Setting classpath:

Class path describes the location where required .class files are available. We can set the class path in the following three ways.

1. By using environment variable “CLASSPATH”.
2. Temporerly at command prompt level. Valid per session.

>Set classpath=E:\gutta.jar;%classpath%;.

1. Temporerly at command prompt level. Valid per one command only.

>Java –cp E:\gutta.jar;C:;D:;. Sample

Jar file

If more no of dependent class files present then it is not recommended to set the class path individually for every class. We have to group all those dependent class files into a single jar file and make that jar file available in the class path.

>jar –cvf gutta.jar \*.\* // to create a jar file.

>jar –xvf gutta.jar // to extract jar file.

>jar –tvf gutta.jar // to display table of contents without extraction i.e without zip.

File I/O

File:

A java file object can represent a file or directory. It is based on unix file system.

Ex: File f=new File(“abc.txt”); // if abc.txt file is already available, then f is pointing to that file. If it is not already available then f simply represents the name of the file, without creating underlying physical file.

Example:

File f=new File(“abc.txt”);

System.out.println(f.exists());

f.createNewFile();

System.out.println(f.exists());

1st run: false, true

2nd run: true, true

A java file can be used to represent a directory also.

File f=new File(“songsfolder”); // if the songs directory is already available then f is pointing to that directory. If it is not already available then f simply represents the name of the directory, instead of creating underlying directory.

Example:

File f=new File(“songsfolder”);

System.out.println(f.exists());

f.mkdir();

System.out.println(f.exists());

1st run: false, true

2nd run: true, true

Note: go through the File class constructors, methods in the api.

FileWriter:

by using FileWriter object, we can write text data (i.e character data) to the file.

FileWriter fw=new FileWriter(String name);

FileWriter fw=new FileWriter(File f);

The above two constructors for overriding existing data.

FileWriter fw=new FileWriter(String name,Boolean append);

FileWriter fw=new FileWriter(File f,Boolean append);

The above two constructors for appending the data.

If the above specified physical file is not already available then these constructors will create that required file.

Note: go through the FileWriter class constructors, methods in the api.

FileReader:

by using FileReader object, we can read text data (i.e character data) from the file.

Note: go through the FilerReader class constructors, methods in the api.

Note: the usage of FileReader & FileWriter is not recommended because of the following three reasons:

* while writing data by using FileWriter, compulsory programmer is responsible to insert new line.
* While reading the data by using FileReader, the no of IO operations will be used. It effects the performance.
* To overcome these problem, we should go for BufferedReader, BufferedWriter classes.

BufferedWriter:

by using BufferedWriter object, we can write text data (i.e character data) to the file.

BufferedWriter bw=new BufferedWriter(Writer w);

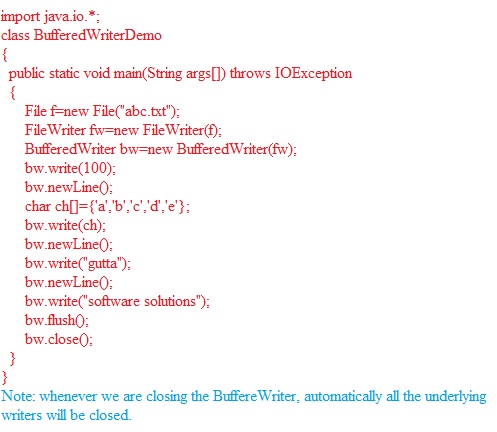
BufferedWriter bw=new BufferedWriter(Writer w, int buffersize);

Note: BufferedWriter can’t communicate directly with the file. It should be communicate via some Writer object.

Q: what are the extra capabilities in BufferedWriter compare to FileWriter?

Writing the data, closing the string, flush the string, adding a new line character.

Example:



BufferedReader:

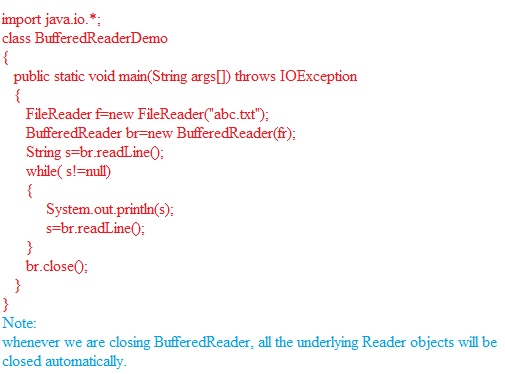
by using BufferedReader object, we can read the text data (i.e character data) from the file.

BufferedReader br=new BufferedReader(Reader r);

BufferedReader br=new BufferedReader (Reader r, int buffersize);

Note: BufferedReader can’t communicate directly with the file. It should be communicate via some Reader object.

Example:



PrintWriter:

The most enhanced type of writer for writing character data to the file is PrintWriter.

By using write() method, we can write only character data to the file. But

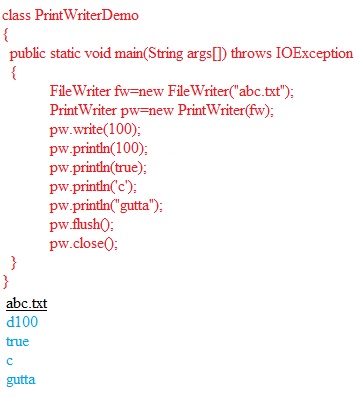
By using print() method, we can write any type of data to the file.

Go through api and have a look at PrintWriter’s methods and constructors.

Q:what is the difference between write(100) and print(100)?

in write(100), the corresponding character i.e D will be added to the file. But

in print(100), 100 value will be added to the file.

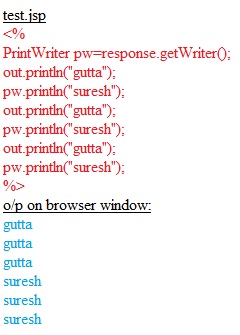


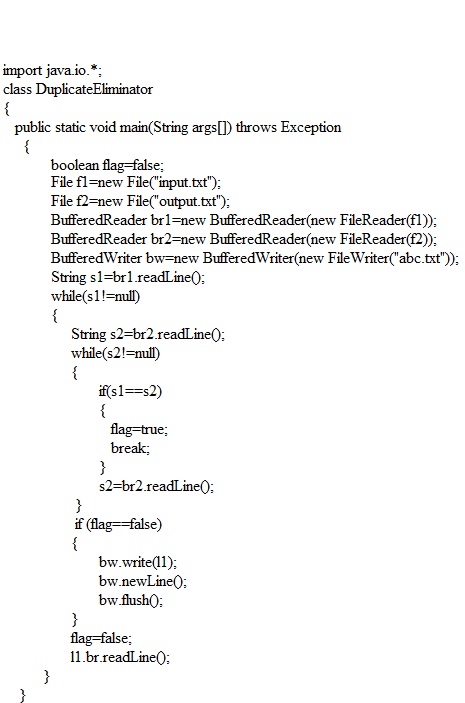
Q: what is the difference between PrintWriter and jspwriter?

In the case of PrintWriter, buffering concept is not available. Hence if we are writing anything to PrintWriter, it will be added immediately to the response header.

But in the case of jspwriter, buffering concept is available, hence if we writing anything to jspwriter, first it will be added to buffer and at last buffer data will be added to respose header.

See the following example for clarity





Flow control:

Flow control describes the order in which statements will be executed at run time.

Flow-control statements

1. Selection statements i.e if-else, switch()
2. Iterative statements i.e while, do while(), for(), for-each loop
3. Transfer statements i.e break, continue, return, try-catch-finally

1.selection statements

If-else statement

Syntax:

if (boolean b)

{

// actions if b is true

}

else

{

// actions if b is false

}

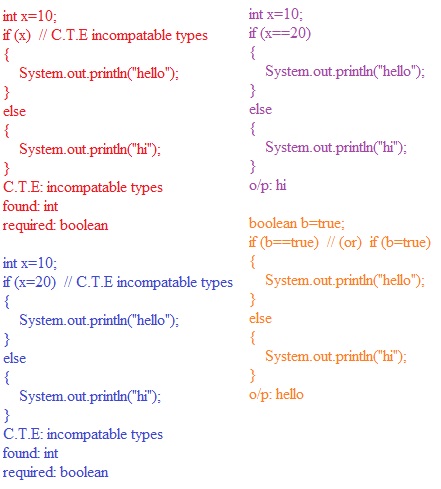
Case (i):

else part & curley braces are optional. Without curley braces only one statement is allowed and that statement should not be declarative statement and we can take empty statement also(;).

|  |  |  |  |
| --- | --- | --- | --- |
| if(true)  System.out.println(“hi”);  The above is correct. | if(true)  int i=10;  the above raises compilte time error. | if(true)  {  int i=10;  }  The above is correct. | If(true);  The above empty statement is correct. |

Case (ii):

The argument to the if statement should be boolean type. By mistake if we are providing any other type we will get compile time error.



Switch statement:

Switch statement is recommended to use if several options are possible.

Syntax:

switch(x)

{

case 1:

--------

--------

case 2:

--------

--------

case 3:

--------

--------

default:

--------

--------

}

Case(i):

curley braces are mandatory.

both case & default are optional.

switch (x)

{

} // this code is correct.

Case(ii):

Within the switch, independent statements are not allowed. i.e every statement should be under case or default.

switch (x)

{

System.out.println(“hello”);

}

// C.T.E saying case or default expected

Case (iii):

The only allowed data types for the switch argument are byte, short, char, int

(Byte, Short, Character, Integer, enum from 1.5). violation leads C.T.E

The case label should be constant expression, otherwise we will get compile time error.

int x=10, y=20;

switch (x)

{

case 10:

System.out.println(“hi”);

Break;

case y: // C.T.E saying case: constant expression required

System.out.println(“hello”);

Break;

}

Note: in the above program, if we declare y as final, then we won’t get any compile time error.

Case (iv):

switch argument can be expression also

int x=10;

switch(x+1)

{

} // this code is valid.

Case (v):

Case label can be expression, but it should be constant expression.

int x=10;

switch(x+1)

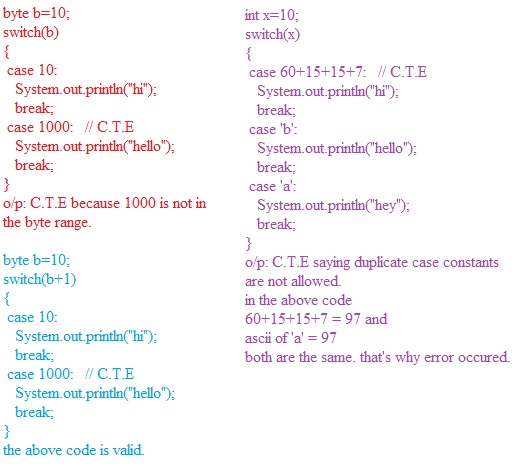
{

case 10:

case 20+30:

} // this code is valid.

Case (vi):

Every case label should be with in the range of switch() argument otherwise C.T.E

Rules for case label:

1. It should be constant
2. Expression also allowed but should be constant expression
3. Duplicates are not allowed
4. Should be in the range of switch argument.

default case:

1. Within the switch, we can take default case any where but recommended to take always at last.
2. Within the switch, if some case is matched then from that statement onwards all the statements will be executed until some break statement or end of the switch. This is called fall-through inside switch.

Example:

Switch(x)

{

case 0:

System.out.println(“0”);

case 1:

System.out.println(“1”);

break;

default:

System.out.println(“def”);

case 2:

System.out.println(“2”);

}

o/p:

if x=0 then 0,1

if x=1 then 1

if x=2 then 2

if x=3 then def, 2

iterative statements:

while loop:

if we don’t know no of iterations in advance, then the best loop is while loop.

Syntax:

While (boolean b)

{

}

Curley braces are optional and without curley braces only one statement is allowed and it should not be declarative statement.

The argument to the while loop is always Boolean, if we take any other type, we will get C.T.E

Example 1:

int x=10;

while (x) // C.T.E

{

System.out.println(“hello”);

}

C.T.E: incompatable statement

Example 2:

while (true)

{

System.out.println(“hello”);

}

System.out.println(“hi”); // C.T.E

C.T.E:unreachable statement

Example 3:

while (true)

{

System.out.println(“hello”);

}

// the above code is correct.

Example 4:

while (true)

{

System.out.println(“hello”);

}; // C.T.E

C.T.E:unreachable statement

Example 5:

int a=10, b=20;

while (a<b)

{

System.out.println(“hello”);

}

System.out.println(“hi”);

The above is valid code.

Example 6:

final int a=10, b=20;

while (a<b)

{

System.out.println(“hello”);

}

System.out.println(“hi”);

C.T.E: unreachable statement

Note: compiler won’t check unreachability in the case of if-else and assert statement.

Ex: if(true)

{

System.out.println(“hi”);

}

else

{

System.out.println(“hello”);

}

The above code correct.

do-while:

if the loop body has to execute at least once, then we should go for do-while.

Syntax:

do

{

// statements

} while(boolean b);

Curley braces are optional. Without curley braces only one statement can be allowed and that statement should not be declarative.

|  |  |  |
| --- | --- | --- |
| do;  while(true);  // the above is valid | do  int i=10; // C.T.E  while (true); | do  while(true);  // invalid |

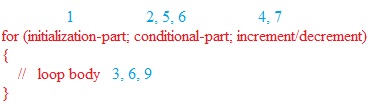
|  |  |  |
| --- | --- | --- |
| do while(true)  System.out.println(“hi”);  While(false);  // valid | do  System.out.println(“hi”);  While(true);  // valid | do  {  int x=10;  }while(true); |
| do  while(true)  System.out.println(“hi”);  while(false);  // valid | do  {  System.out.println(“hi”);  }  while(true);  System.out.println(“hi”);  // valid | do  {  System.out.println(“hello”);  }  while(false);  System.out.println(“hi”);  // valid  o/p is hello, hi |
| Int a=10, b=20;  do  {  System.out.println(“hello”);  }  while(a<b);  System.out.println(“hi”);  // valid  o/p is hello, hello, …….. | int a=10, b=20;  do  {  System.out.println(“hello”);  }  while(a>b);  System.out.println(“hi”);  // valid  o/p is hello, hi | final int a=10, b=20;  do  {  System.out.println(“hello”);  }  while(a<b);  System.out.println(“hi”); //  C.T.E: un reachability |

for-loop:

it is the most commonly used loop

best suitable if we know the no of iterations in advance

syntax: the numbers are the execution flow



Curley braces are optional and without curley braces only one statement is allowed and it should not be declaration statement.

Initialization-part:

This part will be executed only once. In the initialization part, we can declare any no of variables of the same type. But we can’t declare multiple datatype variables on the initialization parth. See the following for clarity.

int i=0, j=0; // valid

int i=0, int j=0; // invalid

int i=0, byte b=0; // invalid

Note: in the initialization part, we can take any java statement, including S.O.P

Example:

int i=0;

for (System.out.println(“hi”); i<2; i++)

{

System.out.println(“hello”);

} o/p: hi, hello, hello

conditional check:

we can take any valid java expression, but it should be of the Boolean type.

Conditional expression is optional and default value is true.

Example:

int i=0;

for (System.out.println(“hi”); ; i++)

{

System.out.println(“hello”);

}

o/p: hi, hello, hello, hello,……..

increment/decrement:

here we can take any valid java statement, including S.O.P

example:

int i=0;

for (System.out.println(“hi”); i<3; System.out.println(“bye”))

{

System.out.println(“hello”);

}

o/p: hi, hello, bye, hello, bye, hello, bye………………..

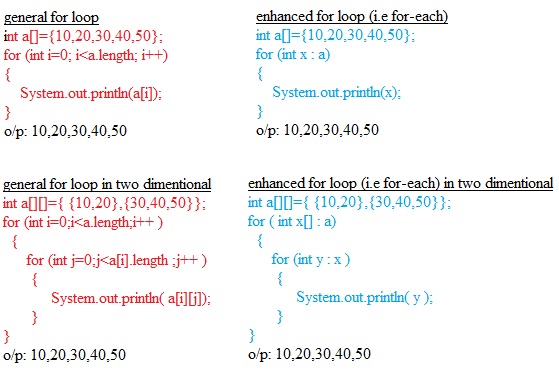
Note:

All the three parts of for loop are independent of each other and optional.

for( ; ; ); // valid, but infinite loop

|  |  |
| --- | --- |
| for (int i=0; true; i++)  {  System.out.println(“hello”);  }  System.out.println(“hi”); // C.T.E unreachable | for (int i=0; false; i++)  { // C.T.E  System.out.println(“hello”);  }  System.out.println(“hi”); // C.T.E unreachable |
| for (int i=0; ; i++)  {  System.out.println(“hello”);  }  System.out.println(“hi”); // C.T.E unreachable | int a=10, b=20;  for ( ; a<b; )  {  System.out.println(“hello”);  }  System.out.println(“hi”);  // valid  o/p:hello, hello, hello…….. |
| int a=10,b=20;  for (; a<b; )  {  System.out.println(“hello”);  }  System.out.println(“hi”); // C.T.E unreachable |

for-each loop:

it is enhancement of for loop, introduced in 1.5 version, best suitable for retrieving the elements of arrays and collections.

even though for-each loop is more powerful, it has one limitation. It is not a general purpose loop. i.e it is applicable only for retrieving elements from arrays & collections.

transfer statements:

break:

we can use break statement in the following places.

1. Within the loops to break the loop based on some condition.
2. Within the switch statement, to stop fall through .
3. Within the labeled blocks to come out of the block based on some condition.

Case (i):

We can write break stmt only in loops i.e in switch, labeled loop, while, do-while, for.

Violation leads compile time error. See the following example

class Test

{

public static void main(String args[])

{

int x=10;

System.out.println(“hello”);

If (x==10)

break; // C.T.E, break is outside switch or loop

System.out.println(“hi”);

} }

Case (ii):

break statement is allowed in labeled block. For labeled block example see the following.

class Test

{

public static void main(String args[])

{

L1:

{

int x=10;

System.out.println(“hello”);

If (x==10)

break L1;

System.out.println(“hi”);

}

System.out.println(“end”);

}

} o/p: hello, end

continue:

We can write continue stmt only in loops i.e in switch, labeled loop, while, do-while, for loop to skip the current iteration and continue with next iteration.

for(int i=10; i<10; i++)

{

if(i%2==0)

continue;

System.out.println(i);

}

o/p: 1,3,5,7,9

case(i):

if we write continue outside the loop, we will get compile time error.

class Test

{

public static void main(String args[])

{

int x=10;

System.out.println(“hello”);

If (x==10)

continue; // C.T.E, break is outside switch or loop

System.out.println(“hi”);

} }

labeled break and continue statements:

in the nested loops, to break or continue a particular loop then should go for labeled break and continue statements. See the following example

L:

for(int i=0; i<3; i++)

{

for(int j=0;j<3;j++)

{

if(i==j)

continue L;

System.out.println(i+“ ----”+j);

}

}

Observe the following outputs:

|  |  |  |  |
| --- | --- | --- | --- |
| break; | continue; | break L; | continue L; |
| 1----0  2----0  2----1 | 0----1  0----2  1----0  1----2  2----0  2----1 | No o/p | 1----0  2----0  2----1 |

Note:

1. int x=0;
2. do
3. {
4. x++;
5. System.out.println(x);
6. if(x<5)
7. continue;
8. x++;
9. System.out.println(x);
10. }while(++x<10);

in the above program, we are not allowed place continue stmt before 6th line. But we can place anywhere after 6th line in the block.

T H E E N D