java.lang package

the most commonly used classes and interfaces for writing any java program are grouped into a separate package and that package by default available to every java program. This package is nothing but java.lang package.

It is not required to import explicitly the lang package classes and interfaces, because by default they are available to every java program. They are

1. Object
2. String, StringBuffer, StringBuilder
3. Wrapper class
4. Auto boxing and Auto un boxing

Object class:

Object is the super class for every java class i.e every class in java is the child class of Object either directly/indirectly.

Object class defines the most commonly used methods which can be applied on any java object irrespective of the class.

Object class defines the following 11 methods.

1. public String toString();
2. public native int hashCode();
3. public Boolean equals(Object o);
4. protected native Object clone() throws CloneNotSupportedException;
5. public final native class getClass();
6. protected void finalize() throws Throwable
7. public final native void wait() throws InterruptedException
8. public final native void wait(long ms) throws InterruptedException
9. public final native void wait(long ms, int ns) throws InterruptedException
10. public final native void notify();
11. public final native void notifyAll();

(i)public String toString():

public String toString()

{

return getClass().getName()+”@”+Integer.toHexString(hashCode());

}

Note: if don’t want the above default implementation, we can override the toString() to provide our own string representation. It is highly recommended to override toString() method too.

(ii)public native int hashCode():

in Object class, hashCode() returns some unique int value which represent address of an object.

Jvm uses hashcode while storing objects in hashtable/hashset/hashmap

Q:what is the relationship between toString() and hashCode()?

In toString() method implementation, we use hashCode() method. See the following

public String toString()

{

return getClass().getName()+”@”+Integer.toHexString(hashCode());

}

Q:is overriding concept applicable for native methods?

Yes, see the following example

class Test

{

int i=10;

public int hashCode() // here we are overriding the Object class’s hashCode() native method.

{

return i;

}

}

(iii)public Boolean equals(Object o):

in Object class, .equals() method is meant for reference comparision. We can override equals method for content comparision.

Q:difference between equals() and == operator?

|  |  |
| --- | --- |
| equals() | == operator |
| We can apply for object references | We can apply for both primitives and object references |
| In Object class, .equals() method is meant for reference comparision. i.e if two references pointing to the same object, then only it returns true. | == operator is always meant for reference comparison only. |
| We can override equals() method for content comparison | We can’t override for content comparison |
| In case of incomparable objects, this method returns false. | In case of incomparable objects, we will get compile time error. |
| For any object reference, t.equals(null) is always false. | For any object reference, r==null is always false. |

Q:relationship between equals() and == operator?

1. If r1==r2 is true then r1.equals(r2) is always true i.e if two objects are equal by == operator then these two objects are always equal by .equals() method also.
2. If r1==r2 is false then r1.equals(r2) need not to be false.
3. If two objects are equal by .equals() method then these mayn’t be equal by == operator.
4. If two objects are not equal by .equals() method then these two always not equal by == operator.

Q:contract between hashCode() and .equals()

1. If two objects are equal by .equals() method and their hashcodes must be equal i.e

If(r1.equals(r2)) is true then r1.hashCode()==r2.hashCode()

1. If two objects are not equal by .equals() method then their hashcodes may or maynot be equal.
2. If hashcodes of two objects are equal then these objects may or mayn’t be equal by .equals() method.
3. If hashcodes of two objects are not equal then these two objects are always not equal by equals() method.
4. to main the above contract between hashCode() and .equals, whenever we are overriding .equals method, compulsory we should override hashCode. Violation leads to no compile time error. But it is not good programming practice.

Ex: 1

String s1=new String(“gutta”);

String s2=new String(“gutta”);

System.out.println( s1== s2 ); // false

System.out.println( s1.equals(s2) ); // true

Ex: 2

StringBuffer s1=new StringBuffer(“gutta”);

StringBuffer s2=new StringBuffer(“gutta”);

System.out.println( s1== s2 ); // false

System.out.println( s1.equals(s2) ); // false

From the above example 1, generally in all wrapper classes and String class, .equals() is overridden for content comparison, hence if content of two string objects is same then .equals() method is always return true.

From the above example 2, in the StringBuffer class, .equals() is not overridden for content comparison, hence Object class’s .equals() has executed which is always meant for reference comparision.

(iv)protected native Object clone() throws CloneNotSupportedException

Creation of exactly duplicate object for backup purpose is called as clone

We can get cloned object by using clone() method of the Object class.

class Test implements Cloneable

{

int i=10;

int j=20;

public static void main(String args[]) throws CloneNotSupportedException

{

Test t1=new Test();

Object o1=t1.clone();

Test t2=(Test)o1;

}

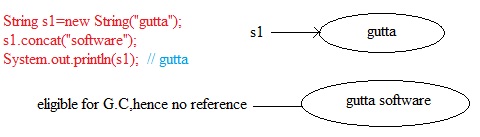
}

(v) public final native class getClass()

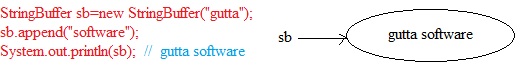
It returns runtime class definition object.

java.lang.String class

case(i):



Note: once we created a String object, we are not allowed to change the content. if we try to change with some changes, then a new String object will be created. This behavior is nothing but immutability of the String behavior.



Note: once we created a StringBuffer object, we are allowed to change the content, if we try to change with some changes, then the current object will be modified instead of creating a new object. This behavior is nothing but mutability of the StringBuffer behavior.

Case(ii):

Ex: 1

String s1=new String(“gutta”);

String s2=new String(“gutta”);

System.out.println( s1== s2 ); // false

System.out.println( s1.equals(s2) ); // true

Ex: 2

StringBuffer s1=new StringBuffer(“gutta”);

StringBuffer s2=new StringBuffer(“gutta”);

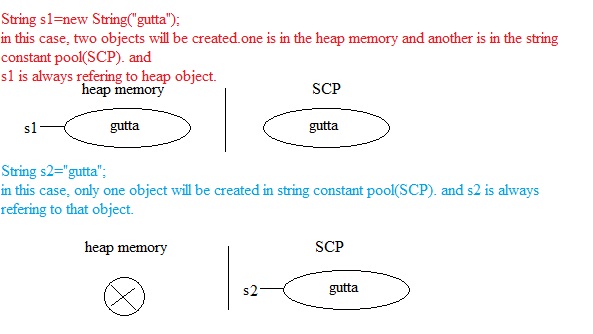
System.out.println( s1== s2 ); // false

System.out.println( s1.equals(s2) ); // false

From the above example 1, generally in all wrapper classes and String class, .equals() is overridden for content comparison, hence if content of two string objects is same then .equals() method is always return true.

From the above example 2, in the StringBuffer class, .equals() is not overridden for content comparison, hence Object class’s .equals() has executed which is always meant for reference comparision.

Case (iii):



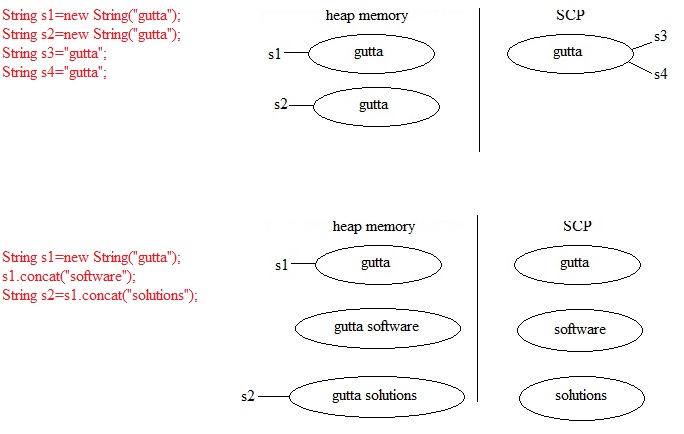
Note:

object creation in SCP is always optional with the required content. If an object is not available then only a new object will be created i.e there is no chance of existing two objects with the same content in SCP. i.e duplicate objects are not allowed in SCP.

Garbage Collector can’t access SCP area. i.e even though objects in the SCP don’t have any reference, they are not eligible for GC.

All SCP objects will be destroyed automatically whenever we shutdown the JVM.

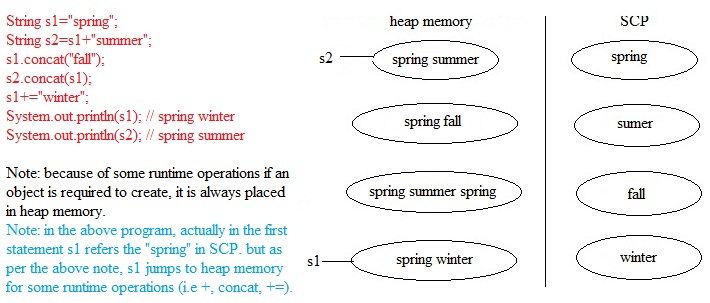
Case(iv):

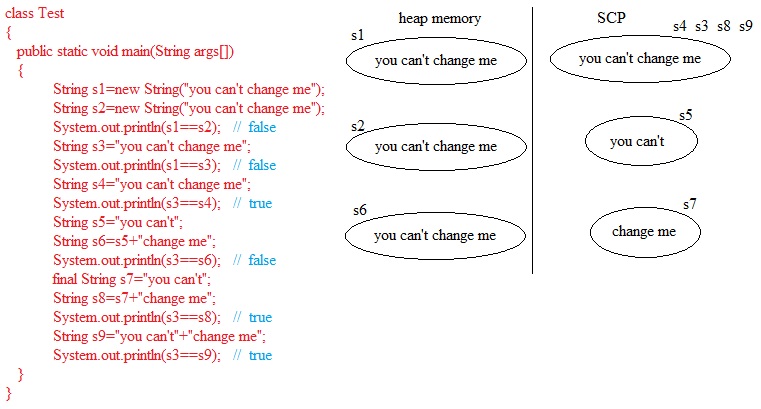


Note:

for every String literal, compulsory one object will be placed in SCP.

because of some run time operations, if an object is required to create, it is always placed in the heap memory. Observe the following example carefully





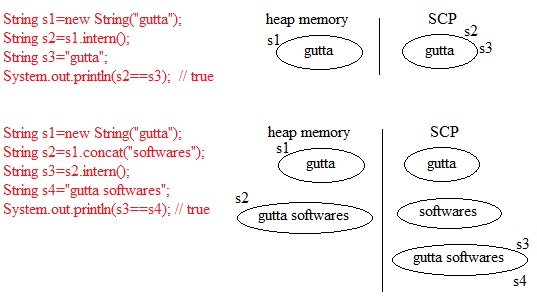
Importance of StringConstant pool:

1. in our program, if a String object will be used repeatedly for every requirement, then instead of creating a separate object for every requirement, we will create one copy in SCP and make it shared by all required references.
2. Instead of creating multiple copies, creating only one copy improves the performance and memory utilization.
3. In several references pointing to the same object by using any reference, if we change the constant , then the remaining references will be effected. To prevent this, SUN people declare String objects as immutable. i.e once we declare String object, we can’t change the content in the existing object. If you try to do any changes on String object, then a new String object will be created.

Interning of the String object:

By using the heap object reference, we can find corresponding SCP object, for this we have to use intern() method.

If the corresponding object is not available in SCP, then intern() method itself will create that object and returns it.



Q: what is the meaning of immutability and explain with an example?

Immutable means we can’t change the immutable object content. If you try to change the content, then a new object will be created with the newly added content. String functionality is an example.

Q:what is SCP?

Q:what are the advantages and limitationsof SCP?

Advantages are saves memory and increases performance.

Limitation is immutability.

Q:other than String object, are there any immutable objects?

All wrapper class objects are immutable.

Q:why is the String class declared as final?

Because, then only no one is allowed to change its implementation(behavior).

Q: is it possible to create our own immutable class?

Yes, it is possible. See the following example

public final class MyImmutable

{

int i;

MyImmutable(int i)

{

this.i=i;

}

public MyImmutable modify(int i)

{

If (this.i==i)

{

return this;

}

else

{

return new MyImmutable(i);

}

}

}

MyImmutable i1=new MyImmutable(10);

MyImmutable i2=new MyImmutable(100);

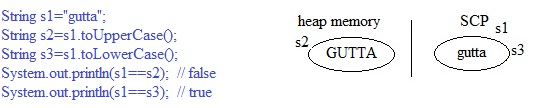
MyImmutable i3=i1.modify(888);

MyImmutable i4=i1.modify(10);

o/p: i2 is separate object, i3 is separate object. But i1, i4 points to the same object.

Note:

when we are applying runtime operations, if the result of any operation is same as the content of actual String object. Then existing object will be returned instead of creating new object. See the following example



java.lang.StringBuffer class

if the content keep on changing frequently, then it is not recommended to use String. because for every modification, a new object will be created internally, and it causes the performance problem.

to handle this kind of requirement, we should go for StringBuffer.

StringBuffer constructors:

StringBuffer sb=new StringBuffer();

it creates an empty StringBuffer object with default initial value 16. Once StringBuffer reaches its max capacity, a new StringBuffer object will be created with new capacity.

new capacity=(current capacity+1)\*2;

example:

StringBuffer sb=new StringBuffer();

System.out.println(sb.capacity());

sb.append(“abcdefghijklmnop”);

System.out.println(sb.capacity());

sb.append(“q”);

System.out.println(sb.capacity());

StringBuffer sb1=new StringBuffer(“gutta”);

System.out.println(sb1.capacity()); // 5+16 i.e capacity=sb1.length+16;

java.lang.StringBuilder class

this is exactly same as StringBuilder except the following differences.

|  |  |
| --- | --- |
| StringBuffer | StringBuilder |
| All methods are synchronized | No method is synchronized |
| StringBuffer object is always thread safe | StringBuilder object is not thread safe |
| Performance is low | Performance is high |
| It is introduced in 1.0 | It is introduced in 1.5 |

Note:

If the content won’t change frequently, then we should go for String.

If the content will change frequently and thread safety is required then we should go for StringBuffer.

If the content will change frequently and thread safety is not required then we should go for StringBuilder.

Method chaining:

in method chaining, all method calls are executed from left to right.

Example:

StringBuffer sb=new StringBuffer();

sb.append(“gutta”).append(“softwares”).insert(2,”xyz”).delete(7,10).reverse().append(“hi”);

System.out.println(sb);

Wrapper classes:

The main objectives(duties) of wrapper classes are

1. to wrap primitives into object form so that we can handle primitives also just like an object.
2. to define several utility functions for the primitives.



constructors of wrapper classes:

Almost of all wrapper classes contain two constructors. One can take the corresponding primitive as argument and the other can take corresponding string as argument.

Integer i=new Integer(10);

Integer i=new Integer(“10”);

If the string is unable to convert to the corresponding number type, we will get Runtime Exception saying NumberFormatException.

|  |  |
| --- | --- |
| Wrapper class | Constructor arguments |
| Byte | byte or string |
| Short | short or string |
| Integer | integer or string |
| Long | long or string |
| Float | float or string or double \* |
| Double | double or string |
| Character | char \* |
| Boolean | boolean or string |

utility methods:

1. valueOf()
2. xxxValue()
3. parsexxx()
4. toString()

1.valueOf() method:

We can use valueOf() method for the creation of wrapper object as alternative to constructor.

form 1:

every wrapper class except character class contains a static valueOf() method for converting string to corresponding wrapper object.

public static wrapperclass valueOf(String s)

eg: Integer i=Integer.valueOf(“10”); // valid

Boolean b=Boolean.valueOf(“true”); // valid

Character ch=Character.valueOf(“a”); // C.T.E

form 2:

every wrapper class including character class contains a static valueOf() method for converting primitive to corresponding wrapper object.

public static wrapperclass valueOf(primitive p)

eg: Integer i=Integer.valueOf(10); // valid

Boolean b=Boolean.valueOf(true); // valid

Character ch=Character.valueOf(‘a’); // valid

form 3:

every integral type wrapper class (i.e Byte, Short, Integer, Long) contain the following valueOf() method.

public static wrapperclass valueOf(String s, int radix) it allows values for the radix are 1 – 36.

Eg: Integer i=Integer.valueOf(“1010”, 2);

conclusion on valueOf():

valueOf()

(string/primitive)---------------------------------🡪(wrapper object)

2.xxxValue():

We can use these methods for converting wrapper object to primitive value.

Every number type wrapper class (i.e Byte, Short, Integer, Long, Float, Double) has this method for converting wrapper object to primitive type.

public byte byteValue();

public short shortValue();

public int intValue();

public long longValue();

public float floatValue();

public double doubleValue();

eg:

Integer i=new Integer(130);

System.out.println(i.byteValue()); // -126

System.out.println(i.shortValue()); // 130

System.out.println(i.intValue()); // 130

System.out.println(i.longValue()); // 130

System.out.println(i.floatValue()); // 130.0

System.out.println(i.doubleValue()); // 130.0

charValue():

Character class contains charValue() method for converting object to char primitive.

public char charValue();

ex: Character ch=new Character(‘a’);

char ch1=ch.charValue();

booleanValue():

Boolean b=new Boolean.valueOf(“gutta”);

System.out.println(b.booleanValue()); // false

Note: totally there are 38 xxxValue() methods. i.e (6\*6+1+1)=38.

conclusion on xxxValue():

xxxValue()

(wrapper object)-------------------------------------🡪(primitive type)

3.parsexxx():

these methods for converting String object to corresponding primitive.

form 1:

every wrapper class except Character class contains the following static parsexxx() method.

public static primitive parsexxx(String str)

ex:

int i=Integer.parseInt(“10”);

double d=Double.parseDouble(“10.5”);

boolean b=Boolean.parseBoolean(“true”);

char c=Character.parseChar(“a”); // it is invalid. C.T.E

form 2:

every integral type wrapper class contains the following parsexxx()

public static primitive parsexxx(String s, int radix); Note: here radix between 2 – 36.

Ex:

int i=Integer.parseInt(“1010”,2); // 10

conclusion on parsexxx():

parsexxx()

(String)---------------------------------🡪(primitive)

4.toString();

form 1:

every wrapper class including Character class contains following toString() method for converting wrapper object to String type.

Ex:

Double d=new Double(10.5);

String s=d.toString();

System.out.println(s); // “10.5”

form 2:

every wrapper class contains a static toString() method for converting primitive value to String type.

Ex:

String s=Boolean.toString(true); // “true”

form 3:

Integer and Long classes contain the following toString() for converting int or long value into specific radix String form.

public static String toString(primitive p, int radix); Note: radix between 2 – 36.

Ex:

String s1=Integer.toString(10,2);

System.out.println(s1); // 1010

String s2=Long.toString(101, 16);

System.out.println(s2); // “a ”

form 4:

Integer & Long classes contain the following toxxxString()

1. public static String toBinaryString(primitive p)
2. public static String toOctalString(primitive p)
3. public static String toHexString(primitive p)

ex:

String s1=Integer.toBinaryString(12);

System.out.println(s1); // 100

String s2=Long.toHexString(100l);

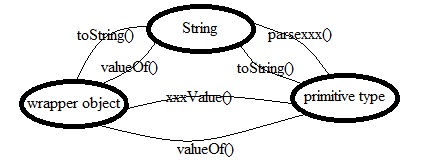
System.out.println(s2); // 64

conclusion on toString():

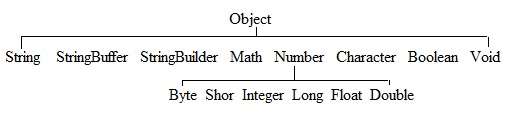
toString()

(wrapper object/primitive)----------------------🡪(String)

dancing between wrapper object, primitive type and String:



partial hirarachy of java.lang package:



Note:

1. some times Void is also treated as wrapper class.
2. in lang package, the final classes are String, StringBuffer, StringBuilder, all wrapper classes.
3. String, all wrapper classes are immutable i.e once we create an object, we are not allowed to change its content.

Auto boxing and Auto unboxing:

Introduction

Until 1.4 version, we are not allowed to provide primitive values in the place of wrapper objects and wrapper objects in the place of primitive values. Compulsory programmer is responsible for converting from primitive to object or object to primitive based on the requirement.

Eg:

ArrayList al=new ArrayList();

al.add(10); // invalid

Integer i=new Integer(10);

al.add(i); // valid

Eg:

Boolean b=new Boolean(“true”);

if(b) // C.T.E saying incompatable types. found: Boolean, required: boolean

{

System.out.println(“check it”);

}

boolean b1=b.booleanValue();

if(b1)

{

System.out.println(“now it is correct”);

}

From 1.5 version onwards, we can provide wrapper object in the place of primitive and primitive in the place of wrapper object. The required conversions automatically performed by the compiler. This concept is nothing but auto boxing and auto unboxing.

Eg:

int i1=new Integer(10); //compiler automatically converts the Integer object to primitive.

Integer i2=20; // compiler automatically converts the primitive to Integer object.

Auto boxing:

Automatic conversion from primitive to wrapper object by the compiler is called auto boxing.

Ex: Integer i=10; Note : it internally calls Integer i=new Integer(10);

Auto unboxing:

Automatic conversion from wrapper object to primitive by the compiler is called auto unboxing.

Ex: int i=new Integer(10); Note: it internally calls int i=Integer.intValue(new Integer(10));

Case (i):

the following is the demo program to show dancing between wrapper object and primitive.

class Test

{

Static Integer i1=10; // auto boxing;

public static void main(String args[])

{

int i2=i1; // auto unboxing

fun(i2);

}

public static void fun(Integer x) // auto boxing

{

int y=x; // auto unboxing

System.out.println(y);

}

}

Case(ii):

class Test

{

Static Integer i1;

public static void main(String args[])

{

int i2=i1; // NullPointerException because it internally calls int i2=Integer.intValue(i1);

System.out.println(i2);

}

}

Case (iii):

class Test

{

public static void main(String args[])

{

Integer x=10;

Integer y=x;

x++;

System.out.println(x); // 10

System.out.println(y); // 10

System.out.println(x==y); // false

}

}

Note: all wrapper objects are immutable i.e once we created a wrapper object, we can’t change its content. If we try to change, then we will get a new wrapper object with modification.

Case(iv):

Integer i1=new Integer(10);

Integer i2=new Integer(10);

System.out.println(i1==i2); // false

Case(v):

Integer i1=new Integer(10);

Integer i2=10;

System.out.println(i1==i2); // false

Case(vi):

Integer i1=10;

Integer i2=10;

System.out.println(i1==i2); // true

Case(vii):

Integer i1=100;

Integer i2=100;

System.out.println(i1==i2); // true

Case(viii):

Integer i1=1000;

Integer i2=1000;

System.out.println(i1==i2); // false

Note:

by auto boxing if any object is required to create then compiler won’t create that object immediately. First it checks whether any object is already created by auto boxing with same content or not. If it is already created, then compiler will gives the existing object instead of creating new object. It saves memory and improves performance. But this rule is applicable for in the following cases only.

1. Byte
2. Short [ -128 to 127 ]
3. Integer [ -128 to 127 ]
4. Long [ -128 to 127 ]
5. Character [ -128 to 127 ]
6. Boolean

Case (i):

Boolean b1=true;

Boolean b2=true;

System.out.println(b1==b2); // true

Case(ii):

Long l1=10l;

Long l2=10l;

System.out.println(l1==l2); // true

Case (iii):

Float f1=10.0f;

Float f2=10.0f;

System.out.println(f1==f2); // true

Case (iv):

Integer i1=128;

Integer i2=128;

System.out.println(i1==i2); // false

overloading with respect to auto boxing, widening, var-args

case (i): widening vs auto boxing

class Test

{

public static void main(String args[])

{

int i=10;

fun(i);

}

public static void fun(long l) // observe that here long is primitive type

{

System.out.println(“widening”);

}

public static void fun(Integer i)

{

System.out.println(“auto boxing”);

}

}

o/p: widening, because compiler always gives the first precedence for widening over auto boxing.

case (ii): widening vs var-args

class Test

{

public static void main(String args[])

{

int i=10;

fun(i);

}

public static void fun(long l) // observe that here long is primitive type

{

System.out.println(“widening”);

}

public static void fun(int … i)

{

System.out.println(“var-arg”);

}

}

o/p: widening, because compiler always gives the first preference for widening over var-arg.

Generally var-arg method executes when no other method matched.

case (iii): var-arg vs auto boxing

class Test

{

public static void main(String args[])

{

int i=10;

fun(i);

}

public static void fun(Integer x)

{

System.out.println(“auto boxing”);

}

public static void fun(int … i)

{

System.out.println(“var-arg”);

}

}

o/p:auto boxing, auto boxing dominates var-arg method.

conclusion on overloading widening, auto boxing, var-arg

while resolving overloaded methods, compiler will always give the precedence in the following order.

1. widening
2. auto boxing
3. var-arg

case (iv):

class Test

{

public static void main(String args[])

{

int i=10;

fun(i); // C.T.E

}

public static void fun(Long l) // observe that here Long is object type

{

System.out.println(“auto boxing”);

}

} o/p: C.T.E

Note: from the above example, we can assume two approaches.

Ok Not ok(because Integer, Long are not child, parent classes)

1. int----------->Integer-------------->Long // finally, it is wrong.

widening auto-boxing

1. int------------------>long-------------------->Long // you may assume that this way is correct. but in java, it is not accepted. i.e

in java, widening, auto-boxing order is not accepted.

in java, auto-boxing, widening order is accepted.

But in second approach, the order is widening, auto-boxing. That’s why it is rejected.

case (iv):

class Test

{

public static void main(String args[])

{

int i=10;

fun(i);

}

public static void fun(Object x)

{

System.out.println(“object”);

}

}

o/p:object, because the flow is like the following

auto-boxing widening

int----------------------->Integer---------------------->Object

Q: which of the following declarations are allowed in java?

1. int i=new Integer(10);
2. Integer i=10;
3. long l=10;
4. Long l=10; // internally Long l=new Integer(10);
5. Long l=10l;
6. Object o=10;

Answer: except 4, all are correct.

­inner classes:

We can declare a class inside another class. such type of clases are called inner classes.

This concept has introduced in jdk 1.1 version as the part of event handling, because of powerful features and benefits of inner classes slowly programmers are using this even in regular coding also.

Without existing outer class object, if there is no chance of existing inner class object, then we should go for inner classes.

Ex:

Without existing car object, there is no chance of existing wheel object. Hence we have to declare wheel class inside the car class.

class Car

{

class Wheel

{

}

}

Ex:

A map is a set of entries. Without existing Map object, there is no chance of existing Entry object. Hence we have to declare Entry interface inside Map interface.

interface Map

{

interface Entry

{

}

}

Based on the purpose and position of declaration, all inner classes are divided into four categories.

1. Normal or regular inner classes
2. Method local inner classes
3. Anonymous inner classes
4. Static nested classes

(i)Normal or regular inner class:

If we declare a class inside another class directly, such type of classes are called as normal or regular inner classes.

Ex:

Sample.java

class Outer

{

class Inner // this is normal or regular inner class

{

}

}

>javac Sample.java

then Outer.class, Outer$Inner.class will be created.

Case (i):

Sample.java

class Outer

{

class Inner

{

}

public static void main(String args[])

{

System.out.println(“outer class main method”);

}

}

>javac Sample.java

>java Outer

Outer class main method

>java Outer$Inner

no such method error: main

Case (ii):

Sample.java

class Outer

{

class Inner

{

public static void main(String args[]) // C.T.E

{

System.out.println(“outer class main method”);

}

}

}

C.T.E saying inner classes can’t have static declaration.

Note: inside inner classes, we are not allowed to declare any static declarations. Hence we can’t declare main method and we can’t invoke inner class directly from the command prompt.

Case (iii):

Accessing inner class code directly from static area of outer class.

class Outer

{

class Inner

{

public void fun()

{

System.out.println(“inner class fun method”);

}

public static void main(String args[])

{

Outer o=new Outer();

Outer.Inner i=o.new Inner();

i.fun();

}

}

}

Case (iv):

Accessing inner class code from instance area of outer class.

class Outer

{

class Inner

{

public void fun()

{

System.out.println(“inner class fun method”);

}

}

public void gun()

{

Inner i=new Inner();

i.fun();

}

public static void main(String args[])

{

Outer o=new Outer();

o.gun();

}

}

Case (v):

Accessing inner class code from outside of outer class.

class Outer

{

class Inner

{

public void fun()

{

System.out.println(“inner class fun method”);

}

}

}

class Test

{

public static void main(String args[])

{

Outer o=new Outer();

Outer.Inner i=o.new Inner();

i.fun();

}

}

Note:

Without existing outer class object, there is no chance of existing inner class object. The relationship between outer and inner classes is not “is a” relationship. “is a” relationship means inheritance.

Case (vi):

All members of outer class(both static and non static) can be accessable from inner classes directly.

class Outer

{

int x=10;

static int y=20;

class Inner

{

public void fun()

{

System.out.println(x); // 10

System.out.println(y); // 20

gun();

}

}

public void gun()

{

System.out.println(“outer class gun method”);

}

public static void main(String args[])

{

new Outer().new Inner().fun();

}

}

Case (vii):

Within the inner class, this always refers current inner class object. to refer outer class object we have to use outer\_class\_name.this.

See the following example for clarity

class Outer

{

int x=10;

class Inner

{

int x=100;

public void fun()

{

System.out.println(x); // 100

System.out.println(this.x); // 100

System.out.println(Outer.this.x); // 10

gun();

}

}

public void gun()

{

System.out.println(“outer class gun method”);

}

public static void main(String args[])

{

new Outer().new Inner().fun();

}

}

Note:

For inner classes, the applicable modifiers are private, default, protected, public, final, abstract, strictfp, static(static nested class)

(ii)Method local inner classes:

Some times we can declare inner class, with in a method also, such type of inner classes are called method local inner classes.

Method local inner classes can be used to define method specific behavior.

The scope of method local inner class is exactly same as method in which it is declared. Hence as the scope is very less, this type of inner classes are very rarely used inner classes.

Ex:

class Outer

{

public void fun()

{

class Inner

{

public void sum(int x, int y)

{

System.out.println(“the sum is: “+(x+y));

}

}

Inner i=new Inner();

i.sum(10,20);

i.sum(100,200);

i.sum(1000,2000);

}

public static void main(String args[])

{

new Outer().fun();

}

}

If we declare inner class within instance method, then we can access all members of(both static and non static) of outer class directly from inner class.

If we declare inner class within a static method, then we can access only static members of outer class from inner class.

From method local inner classes, we can access all members of outer class directly, but we are not allowed to access local variables of the method from its inner class.

If that local variable declared as final, then we can access that variable. see the example

class Test

{

int x=10;

public void fun()

{

int y=20;

class Innter

{

public void gun()

{

System.out.println(x);

System.out.println(y); // C.T.E

}

}

Inner i=new Inner();

i.gun();

}

public static void main(String args[])

{

Test t=new Test();

t.fun();

}

}

C.T.E saying local variable y is access from with in inner class needs to be declared as final.

In the above program, if we declare y as final, then we won’t get any error. Check it out.

case (i):

class Test

{

int x=10;

static int y=20;

public void fun()

{

int z=20;

final int p=40;

class Innter

{

int q=50;

public void gun()

{

--------------1

}

}

}

}

Q:which variables have access at line 1?

Answer: x, y, p, q

Q:which variables have access at line 1 if we declare fun() method as static?

Answer: y, p,q

Q:which variables have access at line 1 if we declare gun() method as static?

Answer: C.T.E, because the only applicable modifiers for method local inner classes are final, abstract.

Anonymous inner classes:

Some times we can declare inner classes without name also, such type of inner classes are called anonymous classes. This type of inner classes are most frequently used.

Anonymous inner classes are divided into the following three categories.

1. Anonymous inner class that extends a class.
2. Anonymous inner class that implements an interface.
3. Anonymous inner class that defined inside method argument.

(i)Anonymous inner class that extends a class

Ex:

class Popcorn

{

public void taste()

{

System.out.println(“salty”);

}

}

class Test

{

public static void main(String args[])

{

Popcorn p1=new Popcor()

{

public void taste()

{

System.out.println(“spicy”);

}

};

p1.taste(); // spicy

Popcorn p2=new Popcorn();

P2.taste(); // salty

}

}

Analysis:

Popcorn p2=new Popcorn(); // here we are creating just an object for Popcorn.

Popcorn p1=new Popcorn()

{

}; // here creating a child class for the Popcorn and for that child class, we are creating an object, with parent reference.this child doesn’t have the name and hence it is anonymous class.

Case(i):

With in anonymous inner classes, we can take child specific method, but these methods are just for internal purpose. From outside anonymous inner class we can call directly.

class Test

{

public static void main(String args[])

{

Thread t1=new Thread()

{

public void run()

{

for (int i=0; i<10; i++)

{

System.out.println(“child thread”);

}

}

};

t1.start();

for (int i=0; i<10; i++)

{

System.out.println(“main thread”);

} } }

o/p:

main thread

child thread

main thread

child thread

.

.

.

(ii)Anonymous inner class that implements an interface:

class Test

{

public static void main(String args[])

{

Runnable r=new Runnable()

{

public void run()

{

for (int i=0; i<10; i++)

{

System.out.println(“child thread”);

}

}

};

Thread t1=new Thread(r);

t1.start();

for (int i=0; i<10; i++)

{

System.out.println(“main thread”);

} } }

o/p:

main thread

child thread

main thread

child thread

.

.

.

(iii)Anonymous inner class that defines inside a method:

class Test

{

public static void main(String args[])

{

new Thread( new Runnable()

{

public void run()

{

for (int i=0; i<10; i++)

{

System.out.println(“child thread”);

}

}

}).start();

for (int i=0; i<10; i++)

{

System.out.println(“main thread”);

} } }

o/p:

main thread

child thread

main thread

child thread

.

.

.

Q:Anonymous inner class vs general class

1. a general class can extend only one class at a time. of course, anonymous class also can extend only one class at a time.
2. a general class can implement any no of interfaces at a time, where as anonymous inner class can implement only one interface at a time.
3. a normal class can extend a class and can implement an interface simultaneously, where as anonymous class can either extend a class or implement an interface, but not both simultaneously.

static nested class:

some times we can declare inner class with static modifier, such type of inner classes are called static nested classes.

static nested class object can exist without existing the outer class object.

inside static nested class, we can declare any static declaration, hence we can declare main method and we can invoke static nested class directly from the command prompt.

Ex:

Sample.java

class Outer

{

static class Nested

{

public static void main(String args[])

{

System.out.println(“nested class main method”);

}

}

public static void main(String args[])

{

System.out.println(“outer class main method”);

}

}

>javac Sample.java

>java Outer

>java Outer$Nested

Ex:

class Outer

{

int i=10;

static int j=20;

static class Nested

{

public void fun()

{

System.out.println(i); // C.T.E non static variable can’t reference from static.

System.out.println(j);

}

}

public static void main(String args[])

{

Outer.Nested n=new Nested();

n.fun();

}

}

Q:comparison between static nested classes and normal inner classes

|  |  |
| --- | --- |
| normal inner classes | static nested classes |
| inner class object is always associated with outer class object. i.e without existing outer class object there is no chance of existing inner class object. | static nested class object is not associated with outer class object, hence without existing outer class also there may be a chance of existing static method class object. |
| inside normal inner classes, we can’t declare static members. | inside static nested class, we can declare static members. |
| In normal inner class, we can’t declare a main method. Hence we can’t invoke directly from the command prompt. | Here we can declare main method, hence we can invoke static nested class code directly from the command prompt. |
| We can access both static and non static members of outer class directly from the inner class. | We can access only static members of outer class directly from static nested classes. |

T H E E N D