Primitive widening uses the "smallest" method argument possible.

Used individually, boxing and var-args are compatible with overloading.

You CANNOT widen from one wrapper type to another. (IS-A fails.)

You CANNOT widen and then box. (An **int** can't become a Long.)

You can box and then widen. (An **int** can become an Object, via Integer.)

You can combine var-args with either widening or boxing.

[**How is ambiguous overloaded method call resolved in java?**](http://javabypatel.blogspot.in/2016/05/ambiguous-method-overloading.html)Overloading.Parameters lists should differ in either,

1. Number of parameters. 2. Data type of parameters. 3. Sequence of data type of parameters.

**advantage :** increases the readability of the program.

**Example of Java API using method Overloading:** 1.  "valueOf",   substring(beginIndex), "println"

**Note:** Method overloading is one of the way through which java supports polymorphism.  
Polymorphishm achieved using method overloading is known as Compile time/Static polymorphism because which method will be invoked is decided at compile time.

**Question 1.**What is the output of below program?

public class OverloadedMethod{

public void test(String str) { System.out.println("String"); }

public void test(Object obj) { System.out.println("Object"); }

public static void main(String[] args) { OverloadedMethod obj = new OverloadedMethod();

obj.test(null); } }

Output: **String**

If more than one method is both accessible and applicable to a method invocation then  
Java compiler uses the set of rules that the most specific method is chosen for invocation.  
In our case above, call to test(null) is suitable for both the test method declared, So in this case most specific method is chosen for invocation.

Compiler check the class hierarchy of method parameter and whichever class is least general one that is the class which is encounter first in bottom up hierarchy, that method is invoked.

Compiler will found String class as the most general class in bottom up inheritance hierarchy, that is why test(String) method is invoked

**Question 2.**What is the output of below program?

public class OverloadedMethod{

public void test(String str) { System.out.println("String"); }

public void test(StringBuffer obj) { System.out.println("Object"); }

public static void main(String[] args) {

OverloadedMethod obj = new OverloadedMethod();

obj.test(null);

}

}

Output: **Compile time error:   The method test(String) is ambiguous for the type OverloadedMethod**  
**Why :** StringBuffer and String class are both at same level in Object hierarchy, So in this case Compiler will not able to resolve which method to invoke and it gives Compile time error.

**Question 3.**What is the output of below program?

class A{} class B extends A{} class C extends B{}

class OverloadedMethod{

public void test(B obj) { System.out.println("B"); }

public void test(C obj) { System.out.println("C"); }

public static void main(String[] args) {

OverloadedMethod obj = new OverloadedMethod(); obj.test(null); } }

Output: **C, because test(null) method call maps to method which contains parameter as class which is lowest in class hierarchy.**

**Question 4.**What is the output of below program?

class OverloadedMethod{

public void test(String obj1, Object obj2) { System.out.println("B"); }

public void test(Object obj1, String obj2) { System.out.println("C"); }

public static void main(String[] args) {

OverloadedMethod obj = new OverloadedMethod();

obj.test(null, null); } }

Output: **Compile time error because method contains parameter in opposite order and none of   
             the test method in this case is more specific and can be resolved.**

**Question 5.**What is the output of below program?

class OverloadedMethod{

public void test(int[] intArr) { System.out.println("int array"); }

public void test(char[] charArr) { System.out.println("char array"); }

public static void main(String[] args) {

OverloadedMethod obj = new OverloadedMethod(); obj.test(null); }

} Output: **Compile time error.**  
int[] is class with name "[I" and superclass of int[] is java.lang.Object.  
char[] is class with name "[C" and superclass of char[] is java.lang.Object.  
                 
 **So both the classes int[] and char[] are at same level to Object class and fall in different**  
**hierarchy, so compiler will not able to resolve as which method to invoke.**  
  
**Note:** int is primitive type in java but int[] is not primitive and it is class which extends Object class.  
you can pass null to int[] because it is object and passing null to int will give compiler error.  
  
You can execute below lines to get the facts,

System.out.println(int[].class.isPrimitive());

System.out.println(int[].class.getName());

System.out.println(int[].class.getSuperclass());

System.out.println(int.class.isPrimitive());

System.out.println(int.class.getName());

System.out.println(int.class.getSuperclass());

Object ref=new int[]{1,2,3}; // valid statement?

Object[] ref=new int[]{1,2,3}; // valid statement?

Output:  false  
[I  
class java.lang.Object  
true  
int  
null  
  
line number 9 is valid statement or not? It is perfectly valid because int[] extends Object and Object is base class.  
line number 10 is invalid statement because int[] extends Object class and not Object[] class.

**Question 6.**What is the output of below program?

class OverloadedMethod{

public void test(Object obj) {

System.out.println("Object");

}

public void test(char[] obj) {

System.out.println("Char");

}

public void test(Integer obj) {

System.out.println("Int");

}

public static void main(String[] args) {

OverloadedMethod obj = new OverloadedMethod();

obj.test(null);

}

}

Output: **Compile time error.**  
char[] is class with name "[C" and superclass of char[] is java.lang.Object.  
              Integer is class and superclass of Integer is Number --> Object.

**So both the classes char[] and Integer are at same level to Object class and fall in**  
**different hierarchy, so compiler will not able to resolve as which method to invoke.**

**Question 7.**What is the output of below program?

class OverloadedMethod{

public void test(long lng) {

System.out.println("Long");

}

public void test(Integer integer) {

System.out.println("Integer");

}

public static void main(String[] args) {

OverloadedMethod obj = new OverloadedMethod();

obj.test(1);

}

}  
Output: **Long**               Because Widening wins over Boxing.  
**Note: Rules that applies for evaluating method call in overloading.**

1. Widening wins over boxing eg. test(10) will call test(long) instead of test(Integer) if both are available.
2. Widening wins over var-args eg test(byte,byte) will call test(int,int) instead of test(byte...x) method.
3. Boxing beats var-args eg test(byte,byte) will call test(Byte,Byte) instead of test(byte...x) method.
4. Widening of reference variable depends on inheritance tree(so, Integer cannot be widened to Long. But, Integer widened to Number because they are in same inheritance hierarchy).
5. You cannot widen and then box. Eg. test(int) cannot call test(Long) since to call test(Long) the compiler need to convert int to Integer then Integer to Long which is not possible.
6. You can box and then widen. Eg. An int can boxed to Integer and then widen to Object.
7. var-args can be combined with either boxing or widening.

**Question 8.**What is the output of below program?

class OverloadedMethod{

public void test(char obj) { System.out.println("Char"); }

public void test(int obj) { System.out.println("Int"); }

public static void main(String[] args) {

OverloadedMethod obj = new OverloadedMethod();

obj.test('a');

}

}

Output: Char  
**What will be the output if there is only test(int obj) method and comment out test(char obj) method?**  
No compile error and output will be "Int", because Java Data type Widening will come in picture.  
  
  
**Java's widening conversions rules are,**  
  
From a byte   ---> short ---> int ---> long ---> float ---> double  
From a short  ---> int ---> long ---> float ---> double  
From a char   ---> int ---> long ---> float ---> double  
From an int    ---> long ---> float ---> double  
From a long   ---> float ---> double  
From a float   ---> double  
**Java's Narrow conversions** **rules are,**  
  
From a byte     ---> char  
From a short    ---> byte ---> char  
From a char     ---> byte ---> short  
From an int      ---> byte ---> short ---> char  
From a long     ---> byte ---> short ---> char ---> int  
From a float     ---> byte ---> short ---> char ---> int ---> long  
From a double  ---> byte ---> short ---> char ---> int ---> long ---> float

**Question 9.**What is the output of below program?

class OverloadedMethod{

public void test(int i) {

System.out.println("Int");

}

public void test(char... c) {

System.out.println("Char varargs");

}

public static void main(String[] args) {

OverloadedMethod obj = new OverloadedMethod();

obj.test('x');

obj.test(97);

}

}

Output:   
**Int**  
**Int**  
  
**Note:**Methods with varargs (...) have the lowest priority.  
           Priority goes as Widening --> Boxing --> Var args.  
  
Logical reason behind Var-args having least priority is varargs were added late in Java API,   
Giving variable arguments a e xtremely low priority is also necessary for backwards-compatibility, otherwise giving high priority to variable argument will mess already written overloaded methods.

**Question 10.**What is the output of below program?

class OverloadedMethod{

public void test(Integer i) {

System.out.println("Integer");

}

public void test(Long l) {

System.out.println("Long");

}

public static void main(String[] args) {

OverloadedMethod obj = new OverloadedMethod();

short s = 10;

obj.test(s);

}

}

Output:  **Compile time error. It is breaking rule "You cannot widen and then box"**  
               For short primitive type to be resolved to either Integer or Long wrapper class, it has to   
first widen short to int and then box it to Integer, which is not possible.

**Question 11.**What is the output of below program?

class OverloadedMethod{

public void test(Character c) {

System.out.println("Character");

}

public void test(Integer i) {

System.out.println("Integer");

}

public void test(Object o) {

System.out.println("Object");

}

public static void main(String[] args) {

OverloadedMethod obj = new OverloadedMethod();

byte b = 10;

obj.test(b);

}

}

Output:  **Object**  
**"You cannot widen and then box" but "You can box and then widen".**  
               obj.test(b) can be resolved to either test(Integer i) or test(Object obj).  
  
               For resolving to test(Integer i), It has to first widen byte to int and then box it to Integer.  
               It is not permissible in java that is **"You cannot widen and then box"**  
  
               For resolving to test(Object i), It has to first box to Integer and then widen to Object.  
               It is permissible in java that is **"You can box and then widen".**and that is why output is    
               "Object".

**Question 12.**What is the output of below program?

class OverloadedMethod{

public void test(char i, Character j) {

System.out.println("method 1");

}

public void test(Character i, Character j) {

System.out.println("method 2");

}

public static void main(String[] args) {

OverloadedMethod obj = new OverloadedMethod();

obj.test('a','a');

}

}

Output:  **Compile Error**  
Why it gave Compile Error?   
  
               Java overloaded method call is resolved using 3 steps,   
**STEP 1:**Compiler will try to resolve call without boxing and unboxing and variable   
                               argument.  
  
**STEP 2:**Compiler will try to resolve call by using boxing and unboxing.  
  
**STEP 3:**Compiler will try to resolve call by using boxing/unboxing and variable argument.   
  
               If call is not resolved by using any of the 3 ways then it gives compile error.  
  
               In our case, method call, test('a','a') is not resolved by using STEP 1 because,  
               In STEP 1, 1st argument of method test(char i, Character j) is mapped but for 2nd parameter   
               it is not able to resolved using STEP 1, so control flow to STEP 2 for method resolution.  
  
               In STEP 2, Compiler tries to resolve call using Boxing and Un-Boxing and found that both   
               method is applicable and valid for call to test('a','a') and that is why Compiler stop here and   
               gave Ambiguous method call error.

**Question 13.**What is the output of below program?

class OverloadedMethod{

public void test(long i, int j) {

System.out.println("method 1");

}

public void test(int i, Integer j) {

System.out.println("method 2");

}

public static void main(String[] args) {

OverloadedMethod obj = new OverloadedMethod();

obj.test(5,6);

}

}

Output: **method 1**  
 The first method has one parameter that requires widening and another that is exact match.   
              The second method has one parameter with exact match and another that requires boxing.   
              Since widening has precedence over boxing, the first method is chosen.  
  
Priority goes as: **Exact Match > Widening > Boxing/unboxing > Varargs**  
  
  
**Question 13.**What is the output of below program?

class OverloadedMethod{

public void test(int... i) {

System.out.println("method 1");

}

public void test(Integer... i) {

System.out.println("method 2");

}

public static void main(String[] args) {

OverloadedMethod obj = new OverloadedMethod();

obj.test(1);

}

}

Output: **Compile time error**  
Why it gave Compile Error?   
               Java overloaded method call is resolved using 3 steps,   
**STEP 1:**Compiler will try to resolve call without boxing and unboxing and variable   
                               argument.  
**STEP 2:**Compiler will try to resolve call by using boxing and unboxing.  
**STEP 3:**Compiler will try to resolve call by using boxing/unboxing and variable argument.   
               If call is not resolved by using any of the 3 ways then it gives compile error.  
               In our case, method call, test(1) is not resolved by using STEP 1 because,  
               In STEP 1,compiler will try data type widening and see whether call is resolved but no in   
               our case.  
               In STEP 2, Compiler tries to resolve call using Boxing and Un-Boxing and see whether call   
               is resolved but no in our case.  
               In STEP 3, Compiler tries to resolve call using Boxing and Un-Boxing along with variable   
               argument and see whether call is resolved. In our case at this step compiler will find both   
               method test(int...) and test(Integer...) is applicable and call  resolved to 2 methods that is   
               why it gave compile error.

**Question 14.**What is the output of below program?

class OverloadedMethod{

public static void main(String[] args) {

System.out.println(null);

}

}

Output: **Compile time error**  
              It gave compiler error because call to println(null) is ambiguous and compiler will not able   
              to decide as which method to invoke among multiple overloaded println() method as none  
              of the class level parameter is more specific.

**Java Language Specification rules for resolving overloading call passes through 3 phases,**

**The first phase (§15.12.2.2) performs overload resolution without permitting boxing or unboxing conversion, or the use of variable arity method invocation. If no applicable method is found during this phase then processing continues to the second phase.**This guarantees that any calls that were valid in the Java programming language before Java SE 5.0 are not considered ambiguous as the result of the introduction of variable arity methods, implicit boxing and/or unboxing. However, the declaration of a variable arity method (§8.4.1) can change the method chosen for a given method method invocation expression, because a variable arity method is treated as a fixed arity method in the first phase. For example, declaring m(Object...) in a class which already declares m(Object) causes m(Object) to no longer be chosen for some invocation expressions (such as m(null)), as m(Object[]) is more specific.  
     
**The second phase (§15.12.2.3) performs overload resolution while allowing boxing and unboxing, but still precludes the use of variable arity method invocation. If no applicable method is found during this phase then processing continues to the third phase.**This ensures that a method is never chosen through variable arity method invocation if it is applicable through fixed arity method invocation.  
  
**The third phase (§15.12.2.4) allows overloading to be combined with variable arity methods, boxing, and unboxing.**

**You may also like to see**

[**Method Overloading - Method Hiding Interview Question-Answer**](http://javabypatel.blogspot.in/2016/04/java-interview-questions.html)

[**How is ambiguous overloaded method call resolved in java?**](http://javabypatel.blogspot.in/2016/05/ambiguous-method-overloading.html)

[**Method Overriding rules in Java**](http://javabypatel.blogspot.in/2016/08/method-overriding-rules-java.html)

[**Interface interview questions and answers in Java**](http://javabypatel.blogspot.in/2016/09/java-interface-interview-questions-in-java.html)

RELATED POSTS :

[Find Smallest and Second smallest e...](http://javabypatel.blogspot.com/2016/11/find-smallest-and-second-smallest-number-in-array.html)

[Find Largest and Second Largest num...](http://javabypatel.blogspot.com/2016/11/find-largest-and-second-largest-number-in-array_9.html)

[Find Largest and Second Largest num...](http://javabypatel.blogspot.com/2016/11/find-largest-and-second-largest-number-in-array.html)

[Find Largest and Smallest number in...](http://javabypatel.blogspot.com/2016/11/find-largest-and-smallest-number-array.html)