Fast Track to Sun Certified Java Programmer (SCJP) 5.0 Upgrade Exam

Copyright © 2006

Ka Iok 'Kent' Tong

Publisher:	TipTec Development
Author's email:	freemant2000@yahoo.com
Book website:	http://www.agileskills2.org
Notice:	All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.
Edition:	First edition 2006

Foreword

Learn the new features in Java SE 5.0

If you'd like to learn the new features in Java SE 5.0 and pass the Sun Certified Java Programmer Upgrade Exam (CX-310-056), then this book is for you. Why?

- It covers all the Java SE 5.0 new features covered in the exam. You don't need to read about the features you already know.
- It is clear & concise. No need to go through hundreds of pages.
- I have passed the exam. So I know what you'll be up against.
- It includes 117 review questions and mock exam questions.
- Many working code fragments are used to show the semantics of the code construct concerned.
- The first 30 pages are freely available on http://www.agileskills2.org. You can judge it yourself.

Target audience and prerequisites

This book is suitable for those who would like to:

- Learn the new features in Java SE 5.0; or
- Take the Sun Certified Java Programmer Upgrade Exam.

In order to understand what's in the book, you need to know the basics of Java. In order to take the exam, you also need to have passed a previous version of the exam.

Acknowledgments

I'd like to thank:

- Helena Lei for proofreading this book.
- Eugenia Chan Peng U for doing the book cover and the layout design.

Table of Contents

Chapter 1

Autoboxing

What's in this chapter?

In this chapter you'll learn about autoboxing.

Autoboxing

In J2SE 1.4 or earlier, you can't directly add say an int to a collection because it needs an Object. You need to wrap it into an Integer object:

```
List list = new ArrayList();
list.add(100); //error: 100 is not an Object
list.add(new Integer(100)); //OK
```

This action of wrapping is called "boxing". In JSE 5.0, whenever the compiler sees that you're trying to assign a primitive value to a variable of a reference type, it will automatically insert the code to convert the primitive value into a wrapper object for you (int => Integer, long => Long, float => Float, double => Double, etc.):

```
List list = new ArrayList();
list.add(100);
```

This is OK in JSE 5.0. The compiler may turn this line into:

```
list.add(new Integer(100));
```

This is called "autoboxing". It not only works for collections, but also for all kinds of assignments:

```
Integer i = 100; //OK
Integer[] a = new Integer[] { new Integer(2), 4, 1, 3}; //OK
g(100);
...
void g(Integer i) {
  ...
}
```

Because it is valid to have:

```
byte b;
b = 100; //100 is an int but it's a constant & within the scope of a byte,
         //so it's converted to a byte.
```

JSE 5.0 will also allow:

```
Byte b;
b = 100; //converted to a byte and then to a Byte (autoboxing)
Byte[] bs = new Byte[] {100, -128, 127}; //do that for each element
```

Auto unboxing

The reverse also works: If you'd like to assign say an Integer object to an int variable, it will be "unboxed" automatically:

```
int i = new Integer(100); //OK
Integer[] a = new Integer[] { new Integer(2), new Integer(5) };
```

```
int j = a[0]; //OK
List list = new ArrayList();
list.add(new Integer(10));
int k = (Integer)list.get(0); //OK
```

Other contexts

Autoboxing and auto unboxing occur not only in assignments, but also in other contexts where the conversion is clearly desired. For example, in an if-statement:

```
if (Boolean.TRUE) { //OK
  ...
}
```

In an arithmetic expression:

```
int i = 10+new Integer(2)*3; //i is 16
Integer j = 10;
j++; //j is 11
```

In a logical expression:

```
if (new Integer(2) > 1) { //OK
  ...
}
```

In a casting:

```
((Integer)100).hashCode(); //OK
((Short)100).hashCode(); //Error: Short is not the right wrapper
```

Autoboxing and method name overloading

Check the code below:

```
class Foo {
  void g(int x) {
    System.out.println("a");
  }
  void g(long x) {
    System.out.println("b");
  }
  void g(Integer x) {
    System.out.println("c");
  }
}
...
new Foo().g(10);
```

What will it print? It will print "a". When the compiler is trying to determine which method it is calling, it will first ignore autoboxing and unboxing. So the first two g() methods are applicable but the third g() is inapplicable. Because the first g() is more specific than the second, it is used. If there were no applicable method, then it would allow autoboxing and unboxing. For example, if the code were:

```
class Foo {
  void g(byte x) {
    System.out.println("a");
  }
  void g(char x) {
    System.out.println("b");
  }
  void g(Integer x) {
    System.out.println("c");
  }
}
...
new Foo().g(10);
```

Then the first two g() methods would be inapplicable because byte and char are narrower than int (the type of the value 10). Then it would proceed to allow autoboxing and find that the third g() would be applicable, so it would be called and would print "c".

Summary

Autoboxing converts an primitive value to a wrapper object. Auto unboxing does the opposite. They work in assignments and other contexts where the conversion is clearly desired.

When finding applicable methods, autoboxing and unboxing are first disabled, so existing code will not be affected by them. If there is no applicable method, they will be enabled to allow more methods.

Review questions

1. Will the following code compile?

```
boolean b = Boolean.TRUE;
if (b) {
  ...
}
```

2. Will the following code compile?

```
if (Boolean.TRUE) {
  ...
}
```

3. Will the following code compile?

```
((Boolean)true).hashCode();
```

4. Will the following code compile?

```
Object[] a = new Object[] { 'a', true, 10.0d, 123, "xyz" };
```

5. What is the output of the following code?

```
public class Foo {
  void g(double d) {
    System.out.println("d");
  }
  void g(Number num) {
    System.out.println("num");
  }
  void g(Object obj) {
    System.out.println("obj");
  }
  public static void main(String[] args) {
    new Foo().g(10);
    new Foo().g(new Integer(10));
  }
}
```

Answers to review questions

1. Will the following code compile?

```
boolean b = Boolean.TRUE;
if (b) {
  ...
}
```

Yes. Autoboxing occurs in the assignment.

2. Will the following code compile?

```
if (new Integer(10)==10) {
  ...
}
```

Yes. Autoboxing occurs in the logical expression.

3. Will the following code compile?

```
((Boolean)true).hashCode();
```

Yes. Autoboxing occurs in a type cast.

4. Will the following code compile?

```
Object[] a = new Object[] { 'a', true, 10.0d, 123, "xyz" };
```

Yes. Autoboxing works for a char, a boolean, a double, an int. For "xyz" there is no autoboxing needed.

5. What is the output of the following code?

```
public class Foo {
  void g(double d) {
    System.out.println("d");
  }
  void g(Number num) {
    System.out.println("num");
  }
  void g(Object obj) {
    System.out.println("obj");
  }
  public static void main(String[] args) {
    new Foo().g(10);
    new Foo().g(new Integer(10));
  }
}
```

For the first call, it will print "d". Because double is wider than int, so the first g () is applicable. At the beginning, autoboxing and unboxing are not considered, so the second and third g() are not applicable. So the first g() is the only applicable method and thus is used.

For the second call, at the beginning autoboxing and unboxing are not considered, so the first g() is inapplicable but the second and third g() methods are. Because the second is more specific than the third, it is used.

Mock exam

1. What is true about the following code?

```
1.  short s1 = 100;
2.  Short s2 = s1;
3.  Integer i = s1;
```

   a. There is a compile error at line 1.

   b. There is a compile error at line 2.

   c. There is a compile error at line 3.

   d. It will compile fine.

2. What is true about the following code?

```
1.  Integer i = 10;
2.  int j = 5;
3.  if (i.compareTo(j) > 0) {
4.    System.out.println("OK");
5.  }
```

   a. It will print "OK".

   b. It will print nothing.

   c. There is a compile error at line 2.

   d. There is a compile error at line 3.

3. What is true about the following code?

```
1.  int i = 10;
2.  Integer j = 5;
3.  if (i.compareTo(j) > 0) {
4.    System.out.println("OK");
5.  }
```

   a. It will print "OK".

   b. It will print nothing.

   c. There is a compile error at line 2.

   d. There is a compile error at line 3.

4. What is true about the following code?

```
1.  class Foo {
2.    void g(int i) {
3.      System.out.println("a");
4.    }
5.  }
6.  class Bar extends Foo {
7.    void g(Integer i) {
8.      System.out.println("b");
9.    }
10. }
11. ...
12. Bar b = new Bar();
13. b.g(10);
```

   a. It will print "a".

b. It will print "b".
c. It will print "ab".
d. It won't compile because the g() in Bar can't override the g() in Foo.

Answers to the mock exam

1. c. Autoboxing will convert a short to a Short, but not to an Integer.
2. a. "j" will be converted to an Integer automatically when it is passed to compareTo().
3. d. "i" will not be converted to an Integer automatically because there is no assignment nor casting there.
4. a. The g() in Bar is not overriding the g() in Foo. It is just overloading the name "g". When determining which method to call, autoboxing is not considered first so only the g() in Foo is applicable and thus is used.

Chapter 2

Generics

What's in this chapter?

In this chapter you'll learn how to use generics and how to create your own.

Using generics

In J2SE 1.4 or earlier, the collection types were like:

```
interface Collection {
  void add(Object obj);
}
interface List extends Collection {
  Object get(int idx);
}
class ArrayList implements List {
  ...
}
```

In JSE 5.0, they have been changed to:

```
interface Collection<E> {
  void add(E obj);
}
interface List<E> extends Collection<E> {
  E get(int idx);
}
class ArrayList<E> implements List<E> {
  ...
}
```

The "E" represents the type of the element of the collection. It is called a "type parameter". When you use them, you need to specify a type as the actual value of E:

```
List<String> list = new ArrayList<String>(); //It reads "a List of Strings"
```

You can imagine that the compiler will generate code like:

```
interface Collection<String> {
  void add(String obj); //can only add String
}
interface List<String> extends Collection<String> {
  String get(int idx); //will return a String
}
class ArrayList<String> implements List<String> {
  ...
}
```

It means that you can call add() and pass a String object but not other types of objects. When you call get(), it will return a String, not an Object:

```
list.add("hello"); //OK
list.add(new Integer(10)); //Error! In J2SE 1.4 it would be OK
String s = list.get(0); //No need to type cast to String anymore
```

The Collection and List interfaces are called "generic interfaces". The ArrayList class is a "generic class". When you provide a type argument, the resulting types such as List<String> or ArrayList<String> are called "parameterized types", while the original types (List, ArrayList) are called "raw types". The act of providing a type argument is called "invocation".

Similarly, the Set interface and its implementation classes are also generic:

```
Set<Integer> set = new TreeSet<Integer>(); //Integer implements Comparable
set.add(new Integer(2));
set.add(5); //OK. Autoboxing.
set.add("hello"); //Error!
if (set.contains(2)) { //It will be true
  ...
}
```

So are Map and its implementation classes:

```
//It has two type parameters
interface Map<K,V> {
  void put(K key, V value);
}
...
//Key is an Integer. Value is a String. Integer implements hashCode().
Map<Integer, String> map = new HashMap<Integer, String>();
map.put(3, "a");
map.put(5, "b");
map.put(4, 2); //Error!
map.put("c", 2); //Error!
String s = map.get(5); //It is "b"
```

Now, the Iterator interface is also generic:

```
interface Collection<E> {
  Iterator<E> iterator();
  ...
}
interface Iterator<E> {
  boolean hasNext();
  E next();
}
```

Therefore, you can iterate a list like this:

```
List<String> list = new ArrayList<String>(); //a List of Strings
list.add("a");
list.add("b");
for (Iterator<String> iter = list.iterator(); iter.hasNext(); ) {
  String s = iter.next(); //No need to type cast
  System.out.println(s);
}
```

Parameterized types are compile-time properties of variables

Consider the code:

```
List<String> list1 = new ArrayList<String>();
List<Integer> list2 = new ArrayList<Integer>();
list1 = list2; //Error
list2 = list1; //Error
if (list1.getClass()==list2.getClass()) { //It will be true!
  ...
}
```

It means the two List objects actually belong to the same class! This is true. When the compiler sees the List interface, it will remove E and change it into Object and use the result as the runtime class (called the "erasure" of the generic class):

```
interface Collection {
  void add(Object obj);
}
interface List extends Collection {
  Object get(int idx);
}
class ArrayList implements List {
  ...
}
```

When you invoke it like this:

```
List<String> list1 = new ArrayList<String>();
```

The compiler will note that the type of the variable list1 is List with the binding of E=String. Later, suppose that there is some code calling add() on list1:

```
list1.add(...);
```

The compiler knows that the type of list1 is List. So it checks the definition of Collection, it finds that the parameter's type of add() is E:

```
interface Collection<E> {
  void add(E obj);
}
interface List<E> extends Collection<E> {
  E get(int idx);
}
```

From the binding in list1, it notes that E=String, so it will check to make sure the method argument is assignment compatible with String. If it is say an Integer, then the compiler will treat it as an error.

Similarly, if you call get() on list1:

```
String s = list1.get(0);
```

From the definition of Collection, the compiler finds that the return type of get() is E. From the binding in list1 it notes that E=String, so it will insert some code to type cast the result to a String:

```
String s = (String)list1.get(0);
```

That is how generic works. The code we imagined before:

```
interface Collection<String> {
  void add(String obj); //can only add String
}
interface List<String> extends Collection<String> {
  String get(int idx); //will return a String
}
class ArrayList<String> implements List<String> {
  ...
}
```

is never generated. At runtime only the raw types exist. The type bindings are properties of variables at compile-time only. They are used during compilation for type checking (as for add()) or to generate type cast code (as for get()). At runtime, the type bindings and the parameterized types no longer exist. Because the "new" operator runs at runtime, new'ing a parameterized type really doesn't make sense:

```
List<String> list = new ArrayList<String>();
```

So, at runtime, what it does is exactly the same as:

```
List<String> list = new ArrayList();
```

However, at compile-time, the <String> is indeed required so that the compiler knows that the type of the expression is ArrayList<String>.

As another example, let's write a class Pair to represent a pair of objects of the same type. You may try:

```
class Pair<E> {
  E obj1; //Compiled as "Object obj1". Doesn't affect anything at runtime.
  E obj2; //Compiled as "Object obj2". Doesn't affect anything at runtime.

  Pair() {
    obj1 = new E(); //Compiled as "new Object()". Affects runtime. Error.
    obj2 = new E(); //Compiled as "new Object()". Affects runtime. Error.
  }
  void setObj1(E o) { //Compiled as "Object o". Doesn't affect runtime.
    obj1 = o;
  }
}
```

Assignment compatibility between parameterized type variables

Obviously the code below won't compile:

```
List<String> list1 = new ArrayList<String>();
List<Integer> list2 = new ArrayList<Integer>();
list1 = list2; //Compile error
list2 = list1; //Compile error
```

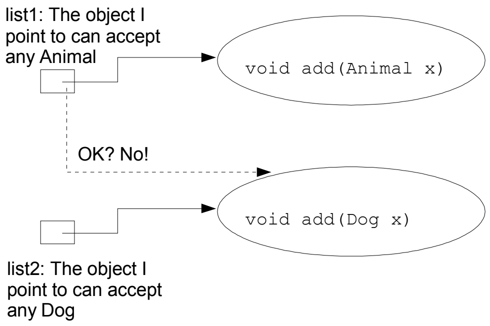
But what about:

```
class Animal {
}
class Dog extends Animal {
}
...
List<Animal> list1 = new ArrayList<Animal>();
List<Dog> list2 = new ArrayList<Dog>();
list1 = list2; //OK?
```

Intuitively, a List of Dog should be a List of Animal. But this is not the case here:

```
List<Animal> list1 = new ArrayList<Animal>();
List<Dog> list2 = new ArrayList<Dog>();
list1 = list2; //OK? No!
list1.add(new Animal());
Dog d = list2.get(0); //Runtime error!
```

For list1, the binding is E=Animal. It means the object it points to can accept any Animal object through its add() method (see the diagram below). For list2, the binding is E=Dog. It means the object it points to can accept any Dog object through its add() method. Obviously we can't just let list1 point to the object pointed to by list2 because that object doesn't accept any Animals, but just Dogs. Our intuition would be correct if the list objects were read-only (e.g., only had a get() method).


list1: The object I point to can accept any Animal
void add(Animal x)
OK? No!
void add(Dog x)
list2: The object I point to can accept any Dog

Therefore, the Java compiler will consider two parameterized types of the same raw type completely unrelated at compile-time and are thus assignment incompatible.

Comparing a List to an array

See if the code below works:

Generic	***Array***

`List<Dog> dogs = new ArrayList<Dog>();`
`List<Animal> animals = dogs;`
	`Dog[] dogs = new Dog[10];`
	`Animal[] animals = dogs;`

As said before, the code on the left hand side won't compile. As a List of Dogs will accept only Dogs but not Animals, you can't assign a List<Dog> to a List<Animal>. But surprisingly, the array version on the right hand side works. It allows you to assign Bar[] to Foo[] if Bar is a subclass of Foo. It means it will allow dangerous code to compile:

```
Dog[] dogs = new Dog[10];
Animal[] animals = dogs; //Dangerous!
animals[0] = new Animal(); //Putting an Animal into a Dog array!
Dog d = dogs[0]; //It is not a Dog!
```

Why it allows such code to compile? Because it has very good runtime checking. When you create an array in Java, it remembers that its element type (e.g., the Dog array above knows its element type is Dog). When you try to put an object into its element, it will perform runtime checking to make sure it is a Dog:

```
Dog[] dogs = new Dog[10];
Animal[] animals = dogs;
animals[0] = new Animal(); //Runtime error
Dog d = dogs[0]; //Won't reach this line
```

Because generics only work at compile-time, they can't perform any runtime checking. Therefore, it takes a strict stance at compile time and forbids you to assign a List<Dog> to a List<Animal>. In contrast, arrays can rely on runtime checking, so it is more relaxed at compile time.

Now, check if the code below compiles:

```
class MyList<E> {
  E[] objs; //Compiled as "Object[] objs". Doesn't affect runtime.

  MyList() {
    E = new E[100]; //OK? No!
  }
}
```

It won't compile. Because an array needs to know its element type when it is created at runtime, the element type must not be a type variable (which doesn't exist at runtime and has been compiled as the type "Object").

Similarly, you can't use a parameterized type as the element type of an array:

```
List<String>[] list; //OK
list = new List<String>[10]; //Compile error
```

Wildcard type

When you provide a type argument for a List, you may write something like:

```
List<?> list1;
```

The "?" represents a type that is unknown to the variable "list1". Such a type is called a "wildcard type". In this case, it is setting the binding E=? for variable "list1" where E is the type parameter in List. This means that list1 is asserting that the object it points to has an add() method that accepts an object of a certain type (unknown to list1) and a get() method that returns an object of a certain type (unknown to list1):

```
class List<?> {
  void add(? obj);
  ? get(int idx);
}
```

It means that we can't add any object to it because we don't know the right type:

```
List<?> list1;
list1.add("a"); //Compile error. It could accept only Integer.
list1.add(100); //Compile error. It could accept only String.
list1.add(new Object()); //Compile error. It could accept only Integer.
list1.add(null); //OK! This is the only thing that you can add to it because
                 // null belongs to any class.
```

We may call get() but we don't know the return type either:

```
List<?> list1;
Object obj = list1.get(0); //Can only declare it as Object
```

If there is another variable "list2" like below, can you assign list2 to list1?

```
List<?> list1;
List<String> list2;
list1 = list2; //OK?
```

To answer that question, you need to note the meaning of list2 having the type of List<String>. It means that list2 is asserting that the object it points to has an add() method that accepts an object of the String type and a get() method that returns an object of the String type:

```
class List<String> {
  void add(String obj);
  String get(int idx);
}
```