New features in Java 7 and 8

Java 7:

1. Can automatically wrap primitives and widen to Object in one step!

For example: Object o = 10 is now legal! It wraps 10 into an Integer and then widens it to Object.

Warning! This feature can lead to very bad programming style, and I do not recommend that you use it. Ever.

2. You don't need to declare generic when using "new" - as long as the Java compiler can figure out what type is needed

Example 1: the type can be determined by the arguments: new LLNode<>("Hello", null)

Example 2: the type can be determined by the type of the variable: LinkedList<String> list = new LinkedList<>();

3. Strings can be used in switch statements

switch (command):

case "quit":

case "enter":

- the switch statement (pre-Java 7) does an == comparison between the switch expression and the case values. So the above code would be comparing

the addresses of the strings. Starting in Java 7, if the values are Strings, it uses the equals() method for the comparison.

Java 8:

1. Interfaces can now contain static methods.

2. Interfaces can now contain "default" non-static methods. Note that a class implementing the interface still inherits the non-static method stubs.

Now, if the inheriting class does not -explicitly- override the method stub, the compiler will add the default method into the class's code and thus -automatically- overriding it.

This is not the same thing as inheriting a method (even though the Java descriptions make it sound as if it is)!

3. Java "lambda" shortcuts for creating anonymous classes.

Java has created two shortcuts that can be used for creating anonymous classes in the special case that the class is implementing an interface that contains a single method stub.

The Java "lambda" shortcut is not a true lambda. (In programming languages, a lambda expression is an anonymous function that can be stored to a variable, input to another function,

returned from a function, etc.)

While the "lambda" shortcut makes it -appear- that we are giving a function as input to a method, we are really giving an instance of an anonymous class

as input to the method.

Example 1: A basic anonymous class with a method that takes one input and no return value:

Recall the ActionListener interface. It has a single method stub actionPerformed(ActionEvent e).

Here is code creating an anonymous class implementing the interface:

b.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

JButton b = (JButton)e.getSource();

b.setText("Click count: " + ++count);

j.pack();

}

});

Here is the "lambda" shortcut. Because there is only one method in ActionListener, there is no ambiguity about which method

is being overridden. All we need to know is what the input parameter name is and what the body of the method is.

Java uses the "->" symbol to indicate that we are using the shortcut.

How does Java know that this is an ActionListener anonymous class? Because that is the parameter type for addActionListener!

b.addActionListener((ActionEvent e) -> {

JButton b2 = (JButton)e.getSource();

b2.setText("Clickercount: " + ++count);

j.pack();

}

);

We can shorten it a little more by noting that the parameter type is also not needed:

b.addActionListener(e -> {

JButton b2 = (JButton)e.getSource();

b2.setText("Clickercount: " + ++count);

j.pack();

}

);

Example 2: An anonymous class with a method that takes one input and returns a value

Here is an anonymous class that implements the Comparator interface:

public Comparator<Name> getComparatorForName() {

return new Comparator<Name>() {

public int compare(Name n1, Name n2) {

return n1.getLastName().compareTo(n2.getLastName());

}

};

}

First, we will do the same "lambda" shortcut as before. Java knows that this is the anonymous class for Comparator because that

is the return type of the method.

public Comparator<Name> getComparatorForName() {

return (Name n1, Name n2) -> {

return n1.getLastName().compareTo(n2.getLastName());

};

}

We can shorten this a little further. Since the body of the overridden method is a single return statement, we can drop the "return" and the curly braces:

public Comparator<Name> getComparatorForName() {

return (Name n1, Name n2) -> n1.getLastName().compareTo(n2.getLastName());

}

Remember, you are not returning a function! You are returning an anonymous class that implements Comparator. The overridden method (this only works if the interface has a single method stub)

takes two Names as input and its body returns the expression after the ->.

Example 3: Using a shorter "method reference" shortcut for an anonymous class.

For this shortcut, you again need to be creating an anonymous class for an interface that has a single method stub. You also need another class that has a method that does exactly

what you want the anonymous class's method to do. Here is another Comparator example:

Suppose our Name class contains a method compareByName:

public class Name {

public int compareByName(Name n1, Name n2) {

return n1.getLastName().compareTo(n2.getLastName());

}

Now, we can use it in a Comparator. Here is the normal way of writing the anonymous class:

Arrays.sort(names, new Comparator<Name>() {

public int compare(Name n1, Name n2) {

return Name.compareByName(n1, n2);

}

});

Note that the anonymous class's body just calls another method. Java 8's new shortcut let's us just specify what method is called inside the method stub:

Arrays.sort(names, Name::compareByName);

This works for non-static methods as well. You just have to place an object of the appropriate type before the ::

Important reminder: This is not changing anything about how Java works! These shortcuts are just programmer conveniences. The Java 8 compiler, when it sees the "lambda" or "method reference"

shortcut replaces it with the "normal" anonymous class creating before compiling the code.