New Features of Java 1.5

By cucu
TopCoder Member

Introduction

This new release of Java brings many significant improvements, not only in the APIs, but also in the language itself. Old code can still run with Java 1.5; but when writing code for this new version you must profit of the new great features that will make your code more robust, powerful and clearer. This feature intends to explain those new features so you can quickly start taking advantage of them.

Auto Boxing and Auto Unboxing

May be something like this happened to you before: you wanted to insert numbers in a list or a map, so you wrote:

```
List l = new ArrayList();

l.add(new Integer (26));
```

You need to use the Integer wrapper because the add method expects an object, and a literal integer is an int, which is not an object. Something similar happens if you have the value stored in a variable.

Then, for retrieving an element, let's say the first one, you can do:

```
int x = ((Integer) l.get(0)).intValue();

System.out.println (x);
```

Beautiful! Actually not very much, that code is relatively hard to read for what it does. This kind of situation arises very often, because primitive types are mainly used to manipulate data but they can't be stored in a collection or passed as a reference. Conversions from primitives to their respective wrappers and vice versa are very common in many Java applications. Since Java 1.5, those conversions are handled automatically by the compiler. That means, if an int value is passed where an Integer or an Object is required; the int is automatically boxed into an Integer. If an Integer is passed where an int is expected, the object is unboxed to an int.

For example, in Java 1.5 this is perfectly valid:

```
Integer value = 26;
```

And has the same effect as writing:

```
Integer value = new Integer (26);
```

For the reverse conversion (unboxing), just use the following:

```
int x = value;
```

That is equivalent to:

```
int x = value.intValue();
```

As you can see, autoboxing and autounboxing makes code look clearer, and it makes code more robust, because the compiler will never box or unbox to a wrong type.

If you box and unbox manually, this could happen:

```
Integer x = new Integer (300);

-

byte y = x.byteValue(); // wrong value: x doesn't fit in a byte

System.out.println(y);
```

This compiles but will give wrong values when x is not in the byte range, and this can be quite hard to debug. If you use Java 1.5 and you let the compiler do its work, you would just write:

```
Integer x = 300;

-

byte y = x;

System.out.println(y);
```

This code is still wrong, but now it doesn't even compile and you'll immediately notice the problem. Even if the compiler cares for those conversions, you must understand that this is being implicitly done in order to avoid doing things like:

```
Integer x=1;

Integer y=2;

Integer z=3;

Integer a;

a= x+y+z;
```

Even if this works the compiler will box the int values to put them in the Integer variables, then it will unbox them to sum its values, and box again to store the Integer variable a, performing much worse than if you would have used int variables. The rule of thumb is that wrappers should be used just when an object representation of a primitive type is needed.

Generics

Probably you've already used the collection classes from Java, which provides classes the means for storing data and applying algorithms on it. For example, the ArrayList class makes easy to use an array that automatically grows as you need. But you might need an array of integers, strings or a Person class you've defined. The algorithms for working on any of those kinds of data are identical, so it won't make sense to rewrite the code for each data type. In previous versions of Java, they solved this by making the collection classes take Object as their elements; and because any object inherits from that class, any instance could be passed to the collections.

For example:

```
List list = new ArrayList();

list.add("hello");



-
```

```
String s;

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

Note than when retrieving elements, very often a cast must be used because the returned type is an Object, and the stored object is probably from another class.

The main problem with this approach is that you're "loosing" the type of your objects and then doing a cast because you know beforehand what type they're. But this is very error prone, because if you do the wrong cast, the program will run but give a runtime error. Java 1.5 introduced Generics, which make this kind of code much clearer and type safe.

The above code will be written as:

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

list.add("hello");



String s;

s = list.get(0);
```

The class List is now generic, that means that it takes parameterized types; in that case the type is String. With List<String> you're saying to the List class that this instance will work with the type String. Then, the constructor is also called with the parameter type in order to make the instance be of that type. Now the cast is removed, because the get method returns a String, as a consequence of the list declaration. That way, you work with your instance as if it were specifically designed to work with Strings, making you save casts that are risky.

Also it's possible, and sometimes very useful, to have more than one type as parameter; for example the maps needs two types, one for the key and one for the values:

```
Map<String, Integer> map = new HashMap<String, Integer>();



map.put("Bill", 40);

map.put("Jack", 35);



System.out.println(map.get("Jack"));
```

This forces the keys to be Strings and the values to be Integers.

Now, let's see how we can declare a generic class, doing a class that holds any kind of object:

```
public class Holder<T> {

    private T value;



    void set(T value) {

        this.value = value;

    }



    public T get() {
```

```
        return value;

    }

}
```

The parameterized types are right after the class name, enclosed between angle brackets. If using more than one, they must be comma separated, for example "<K,V>" is used for Map class. Then, we can use T almost (later we'll see the differences) as if it were a class, even if we don't know which one is it. This class must be specified when declaring instances, for example we could use the Holder class as follows:

```
Holder<String> h = new Holder<String>();

h.set("hello!");

System.out.println(h);
```

In the first line we are declaring h as an instance of Holder<String>, so we can think of T as being a String and read the Holder class doing a mentally search and replace. As you can see, the set method takes an argument of type T (in this case String), so if another class is used instead, it will generate a compilation error. The method get returns an object of type T (again String), so we don't need to cast to String when calling it. The way the compiler does that is through erasure; that means that it actually compiles generic classes as if they work with Object instances and then it applies casts and do checks for type safety. This is different as how C++ works (it generates one copy of the class for each different type instantiated) and has some consequences on its use. I said before that T should be treated *almost* as if it were a class, because there are things that you can't do, due to the erasure procedure.

For example,

```
value = new T();
```

will not compile even if it seems to be right and quite useful. The problem is that the compiler doesn't have the information of which type is T at compile time, so it can't create the instance. For the same reason, you can't create an array of T's:

```
value = new T[5];
```

will also give a compiler error. Let's make a class for handling complex numbers using generics, so it can work with different numeric types:

```
public class Complex<T> {



    private T re;

    private T im;



    public Complex(T re, T im) {

        this.re = re;

        this.im = im;

    }



    public T getReal() {

        return re;

    }
```

```
    public T getImage() {
        return im;
    }

    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

The constructor will take two variables of type T, the real and imaginary parts.

We can use that class as follows:

```
Complex<Integer> c= new Complex<Integer>(3,4);
System.out.println (c);
```

Getting as output: "(3, 4)"

Notice that Integer is used because only classes can be parameters; primitive types are not allowed. However, thanks to autoboxing we can pass int parameters to the constructor to make life easier; if not we should do:

```
Complex<Integer> c= new Complex<Integer>(new Integer(3), new Integer(4));
```

We could do some other things with the Complex class, for example:

```
Complex<String> c= new Complex<String>("hello","world");
```

But hey, this is not the idea of a complex number! We wanted to use generics so it could hold different types of numbers. Let's leave that aside for a brief moment to add a method to calculate the modulus:

```
public double getModulus() {
       return Math.sqrt(Math.pow(re, 2) + Math.pow(im, 2));
}
```

But this doesn't even compile! However this behavior seems reasonable; if it would have compiled, how would it have solved the modulus of the last complex instantiated?

We need to get a numeric value from re and im, and this could be done using doubleValue() method:

```
return Math.sqrt(Math.pow(re.doubleValue(), 2) +
Math.pow(im.doubleValue(), 2));
```

We're nearer now, but the compiler still doesn't like it-how can it know that re and im have the doubleValue method? If we pass a String, how would it solve that? The answer to all those questions is to promise the compiler that re and im will be numbers; that is, they'll be instances of classes that inherit from Number. To do that, you just have to modify the declaration of the class to be:

```
public class Complex<T extends Number> {
```

That way, you're saying that T must be a Number or a subclass. This is called a bounded type. Now, you won't be able to compile Complex<String> because String is not a Number subclass. Because Number class defines the method doubleValue() - and thus it is defined for all its subclasses - you can use this method on re and im variables, as well as you can call any method defined in Number or its superclasses.

Let's go further and define a method to compare the vector length (i.e. its modulus) with other vector:

```
public boolean isLarger(Complex<T> c) {

  return getModulus() > c.getModulus();

}
```

This can be used as follows:

```
Complex<Integer> x= new Complex<Integer>(3,4);

Complex<Integer> y = new Complex<Integer>(4,5);



System.out.println (x.isLarger(y));
```

And it works as expected. However, we might also want to do:

```
Complex<Integer> x= new Complex<Integer>(3,4);

Complex<Double> y = new Complex<Double>(4.2,5.8);



System.out.println (x.isLarger(y));
```

And this doesn't even compile, because x is a vector of Integer and it expects the same type in the isLarger method. However, we know that this is a valid operation, because even if they real and imaginary parts are of different types, we still can compare their modulus. What we need now is to use wildcards to tell the isLarger method that we really don't care for the type of complex we receive, and this is very simple to do:

```
public boolean isLarger(Complex<?> c) {
```

The interrogation sign stands for the wildcard, meaning any type. Because the Complex type is already bounded to Number, actually the type can't be other than a Number. Wildcards can also be bounded in the same way we bounded T to be a Number.

You may have noticed that at the beginning of the section I used the ArrayList class as follows:

```
new ArrayList<String>();
```

But ArrayList is an old class, at least older than Java 1.5, so how does it comes that now it takes a type? Is it in another package? Is the same class and they've broken backwards compatibility? The answer is that is the same class as in previous versions of Java; but this new version wouldn't be well received if it wouldn't be backwards compatible, so even if it can used as a generic class, omitting the type will result in a raw class, which works over the Object class as the previous Java versions. That way, the old code won't break, but new code should use the parameterized form to be type safe. When using the raw type, the Java compiler issues a warning about that. Generic classes can be inherited or can inherit from other generic or non generic classes; all the possibilities are valid.

For example, you could have another complex class that has some extra functionality and declare it as:

```
public class BetterComplex<T extends Number> extends Complex<T> {
```

Note that T should be declared at least to be bounded to Number, otherwise it won't compile because Complex is not guaranteed to have its parameter bounded to Number. It could also be bounded to a subclass of Number, but not to anything else. The parameter list could have more than the T parameter in BetterComplex; however you should always give Complex exactly one type. You can also make a non generic class inheriting from a generic class.

For example if you define some kind of Complex that you know their values are always of type Double:

```
class BetterComplex extends Complex<Double> {
```

That way, you don't even need to specify a type (actually you can't) to BetterComplex type.

The For-Each loop

A task that is done often is to iterate through a collection, and do something with each element. For example, this shows all the elements in an array:

```
String fruits[]= {"apple", "orange", "strawberry"};



      for (int i = 0; i < fruits.length; i++) {

            System.out.println (fruits[i]);

      }
```

We could also think of a method to show all the elements in a list:

```
public void Show(List l) {

     for (int i = 0; i < l.size(); i++) {

          System.out.println (l.get(i));

     }

}
```

This code seems right; however it could be very inefficient. Surprised? Try to guess why - And if not, just run the following:

```
List l = new LinkedList();

for (int i = 0; i < 10000; i++) {

      l.add("number " + i);

}

Show(l);
```

You'll have some time to think why it is inefficient while you wait for it to finish running. The reason is that a linked list doesn't have random access to its elements, so when you ask for an element, the linked list will sequentially search your element. For getting the 10000th element, the entire list will be iterated. So, the problem with that approach is that depending on the list implementation you could get an algorithm of order O(n^2) whereas an algorithm O(n) is easily obtained.

One way of making this algorithm work in O(n) independently of the list implementation is using iterators. This is the new version of the show method:

```
public static void ShowFast(List l) {

     for (Iterator it = l.iterator(); it.hasNext();) {

          System.out.println (it.next());

     }
```

```
}
```

When you ask for an iterator of the collection using the method iterator(), a reference to the beginning of the list is retrieved. Then, the hasNext() method returns whether there are still more elements to come, and it.next() does two things: it advances the reference to the next element in the collection and retrieves the current element. This gives the LinkedList the opportunity to iterate through the list in O(n). For the moment, all that can be done in earlier versions of Java. Even if the above code works, it's not nice and quite error prone. For example, if in the hurry you call again to it.next() in the for block in order to retrieve again the element, you'll get half the list in one place and half in another. Of course that this can be solved storing the value of it.next() in a local variable, but Java 1.5 brings a nicer and safer way to do the same: the for-each loop.

The method is written this way:

```
public static void ShowFastAndNice(List l) {

     for (Object o : l) {

          System.out.println (o);

     }

}
```

You can read the for sentence as "for each Object o in l". As you can see, the for keyword is still used, but now it takes the syntax:

```
for(type iteration_var : iterable_Object) code
```

The break keyword can be used in the same way as in the regular for. Observe that the type is mandatory; for example this won't compile:

```
Object o;

   for (o : l) {

         System.out.println (o);

   }
```

With the compiler forcing the iteration variable to be in the iteration block, a little flexibility is lost, because that variable can't be used after exiting the for, a practice that is common, for example, if the for is broken when something is found. However these kinds of practices are not very clear and robust, so the compiler is making you to write better code, even if it might require an additional flag. The iterable_object is any object that implements the Iterable interface, which only defines the method Iterator<T> iterator(). The type must be compatible with the type returned by the next() method of the iterator.

Arrays can also be used, so the loop of the first example in this section could be written as:

```
for (String fruit : fruits) {

    System.out.println (fruit);

}
```

To make your own types compatibles with the for-each loop, you must implement that interface and return the appropriate iterator, that often is the object itself, who is also implementing Iterator. You should use this form of for as much as you can; however there are some cases where it is not possible or straightforward:

- When you will remove some elements in the collection; in that case you need to have the iterator to call the remove method.
- When you're iterating through more than one collection at the same time, for example you have an array of names and one of values and you know that value(i) belongs to name(i)

Variable-Length Arguments

Imagine that you're programming an application that repeatedly needs to know the maximum of some integer variables. You probably will want to put the code in a method.

For example, you might do:

```
int x0, x1;

-

int z = max(x0, x1);
```

And define a method max that takes 2 arguments and returns the bigger of the two. So, you code that method and happily continue programming until you find that sometimes you need to calculate the maximum between x0, x1 and x2, and sometimes between 10 variables. One approach would be to overload the max method, defining it with different quantities of parameters, which is laborious and ugly.

A better approach is to receive an array, so your method max could be:

```
public int max(int []x) {

  int maxValue = x[0];

  for (int value : x)  {

      if (value >  maxValue) {

          maxValue = value;

      }

  }

  return maxValue;

}
```

The method uses the for-each to get the maximum element in the array. Note that if an empty array is passed, an ArrayIndexOutOfBoundsException will be thrown; you could check that case and throw another exception to explain that it can't be called with an empty array. Now that we have a nice method that takes any number of arguments, let's use it:

```
int x0, x1, x2, x3;

-

int y = max(new int[] {x0, x1});

int z = max(new int[] {x0, x1, x2, x3});
```

Even if it works, it makes the code very unclear, but with Java 1.5, you can have your cake and eat it, too! With variable-length arguments you can make the method take any number of arguments and call it in a straightforward way. The variable-length arguments are specified using three dots before the variable name.
br> For example, the above method will be written now as:

```
public int max(int ...x) {

  int maxValue = x[0];

  for (int value : x)  {

      if (value >  maxValue) {

          maxValue = value;

      }

  }
```

```
  return maxValue;

}
```

The only difference is in the signature; observe that x is treated as a normal array. The advantage is when you have to call that method. Now you can use:

```
int y = max(x0, x1);

int z = max(x0, x1, x2, x3);

int error = max(); // it will throw an exception
```

The compiler will internally do something very similar as we did before: create an array with all the values and pass it to the method that will receive them as an array.

You're still able to call the method with an array of integers, which can be useful if you need to build it programmatically, or if you want to upgrade your methods to variable arguments without breaking compatibility.

When using variable-length arguments, zero arguments are permitted, which sometimes is an advantage, and others, as in that case, is something that you don't desire.

You might be tempted to write the method declaration in the following way:

```
public int max(int y, int ...x) {

// get the maximum between y and the elements of the array x

}
```

This will force the user to pass at least one parameter. Doing this brings a complication: if you want to pass an array, you have to pass its first element as the first parameter and the rest as the second parameter, and that code will look really bad. Normal and variable-length parameters can be combined, but not any combination is valid. The variable-length argument must be the last parameter, which implies that it's not possible to have more than one variable-length argument in a method declaration. Sometimes there can be ambiguity between overloaded methods that take variable-length arguments.

For example, consider the following declarations:

```
public int method(int ...x) {

// do something

}



public int method(String ...x) {

// do something

}
```

I'm sure you can deduct which method will be called in those statements:

```
method(3);

method("3");

method(1, 2, 3);

method("hello", "world");
```

But, you can't know which method will be called if I do:

```
method();
```

Both method declarations match, so you can't know, and neither the compiler does, so the above statement will not compile. You'll have an error message saying: "reference to method is ambiguous,". The ambiguity also happens if you declare:

```
public int method(int ...x) {

// do something

}



public int method(int y, int...x) {

// do something

}
```

And try to call the method with one or more arguments; those declarations will probably be useless. Variable-length arguments make code look clearer in many common situations. However, there are situations where they can't help: for example a method that takes names and values for the names, could be written as:

```
public void multiple(String names[], int values[]) {

// do something with names and values.

}
```

But it can't be translated to variable-length arguments, because this won't compile:

```
public void multiple(String -names, int -values) {

   /-

}
```

There's no way to use variable-length arguments in this example, but don't worry; just use the arrays in the old way.

Enumerations

Sometimes you need to manage information that doesn't fit well in the existent types. Imagine for example a method that needs to receive which programming language a coder uses. The method could be:

```
public void doSomething(String handle, String language) {

   if (language.equals("Java")) {

        System.out.println ("Hope you already know 1.5!");

   }

   -

}
```

And then, it will be called using, for example

```
doSomething("abc","Java")
```

This has lots of problems: you could misspell the name, you might not know all the available names, you could pass an inexistent language and so on. A better approach is to use constants, for example you could write the following code:

```
public class Languages {

    public static final int JAVA = 1;

    public static final int CPP = 2;

    public static final int CSHARP = 3;

    public static final int VBNET = 4;

}
```

Then, your method doSomething will be:

```
public void doSomething(String handle, int language) {

   if (language == Languages.Java) {

       System.out.println ("Hope you already know 1.5!");

   }

   -

}
```

And your call will be:

```
doSomething("abc",Languages.JAVA)
```

This is better; at least you're sure that any misspelling is caught at compile time; however if you do:

```
doSomething("abc", 6)
```

You're giving the method a code for a language that doesn't exist. Java 1.5 introduced enumerations, that is something that exist in many older languages; however Java 1.5 has enumerations far more powerful than the ones found on those languages. Let's first see how the above example will be done.

We define the enumeration type as following:

```
enum Language { JAVA, CPP, CHSARP, VBNET }
```

Now we have a new type called Language and we can define variables of that type, and assign any of the declared values:

```
Language lang;

lang = Language.JAVA;
```

Note that Lanuage.JAVA is like an instance of Language, which is similar to a class, but can't be instantiated. The method will be coded now as:

```
public void doSomething(String handle, Language language) {
```

```
    if (language== Language.JAVA) {
        System.out.println ("Hope you already know 1.5!");
    }
}
```

And the call will be:

```
doSomething("abc", Language.JAVA)
```

Now, it's impossible to pass anything else than the defined values, making the method safer. But Java doesn't stop here. While in some languages the enumerations are a bit more than a collection of integer constants, in Java many nice additional features are provided.

Try this one:

```
System.out.println (Language.JAVA);
```

This will print "JAVA", so that you don't have to do a map from the enumeration values to its names; Java already stores the name of the constant. You can also know all the available values for a type using values() method, that returns an array of the enumeration type, and can be used as follows:

```
System.out.println ("Available Languages:");
for(Language l : Language.values())
   System.out.println(l);
```

This will print the list of the four available languages. Now, imagine that the user is asked for the language he's willing to use, this could be done using a "combo box" where the values are retrieved as above, so that if new languages are added, they automatically appear in the combo box as soon as you add the constant in the enumeration. If you want to call the method doSomething, you'll notice that a Language constant is needed, but we have a string constant from the combo box (or may be some free text input). To translate it, you have a very easy way: the method valueOf(String str) that returns the enumeration constant corresponding to that name, or throws an IllegalArgumentException if not found. You could do:

```
doSomething("abc", Language.valueOf(selectedLang));
```

where selectedLang is the String variable that holds the user selection. Now, imagine that our method doSomething needs to know the extension of the file for saving it. We could do:

```
String extension;
switch (language) {
  case JAVA: extension = ".java"; break;
  case CPP: extension = ".cpp"; break;
  case CSHARP: extension = ".cs"; break;
  case VBNET: extension = ".vb"; break;
  default: throw new IllegalArgumentException
              ("don't know that language extension");
}
// do something using extension variable
```

This example shows that the switch sentence can be used with enumerations. Notice that the languages in the switch are referred without specifying the enumeration type. If you try to compile the above code without the default in the switch, you'll got an error saying that extension might not be initialized. But you're sure it is!! language variable is bounded to be one of the four defined constants, so there's no way to escape! You might be tempted to just initialize the extension to an empty string or null to shout the compiler's mouth, and even if it will perfectly work for the moment, is not the best for being extensible. If other language is added and you forget to add the case statement, the extension variable will be null or empty string, and the error might be much more complicated to find than if you throw an exception as we did. But this solution is still not the best; as we've been talking, you could easily forget to declare the case statement, and you might have many "maps" as the above in different parts of code, making it hard to maintain. Because the extension is linked to the language itself, why not store the extension itself on the constant? An enumeration is actually a special kind of class, and it can have methods and attributes, so you can do:

```
enum Language {

        JAVA   (".java"),

        CPP    (".cpp"),

        CSHARP (".cs"),

        VBNET  (".vb");



        private String extension;



        Language(String ext) {

            extension = ext;

        }



        public String getExtension() {

            return extension;

        }

}
```

Now the constant definition must match the only available constructor. The constructor can't be declared as public, you'll get a compile error because enums can't be instantiated. You could overload the constructor to provide many ways to initialize the constants, in the same way you would do in a normal class. Now, our method doSomething will simply do:

```
String extension = language.getExtension();

// do something using extension variable,

// that actually is not even needed now.
```

The code is more robust now, because when adding a new language is impossible to forget to initialize the extension; you won't be able to compile if you don't specify it. And as a plus, you are declaring the extension in the same place as the language, so you don't need to search all the code for switch'es to add the extension.

In conclusion, Java provides a new kind of enumerations that is very powerful because the constants are more like a singleton instance than an integer variable, so more information can be stored in the constant; and even singletons won't be so good to use as constants because enums provide extra functionality like listing all the constants for a type and mapping from a constant to its string representation and vice versa.

Annotations

Annotations, or metadata, introduce a new dimension in your code. They enable you to embed information in your source code. That information does nothing by itself; the idea is that another program can use it. For example, when writing unit tests, the JUnit framework needs to know which methods are you willing to test; and it does that using reflection to get all the methods whose name starts with "test", as well as to look up for methods "setUp", "tearDown" and so on. Annotations can help here, marking that a method is a test method, that an exception is expected and so

on. Actually, it's quite probable that in a near future JUnit will implement this, because NUnit is already using the analogous idea in .Net platform. Now, a method that tests if a constructor throws NullPointerException looks like this:

```
public void testNullConstructor () {
   try {
       new TestedClass(null);
   } catch (NullPointerException e) {
   }
}
```

Using annotations, it could look like:

```
@TestMethod
@ExpectedException(NullPointerException.class)
   public void nullConstructor () {
   new TestedClass();
}
```

As you can see, there are two annotations before the method that are recognized by their starting "@". The name of the first annotation is TestMethod and it doesn't take any parameters. The second annotation (ExpectedException) takes one parameter. Now thay you understand what they can be used for, let's see how they're declared and accessed.

An annotation is based on an interface, but it can just have method declarations that take no parameters and the return type must be one of the primitive types, String, Class, an enum type, an annotation type, or an array of one of the preceding types. For example, this is an annotation declaration:

```
@interface Note {
   String author();
   String note();
   int version();
}
```

This annotation will be called, for example, in the following way:

```
@Note(author="TCSDEVELOPER", note="Working fine", version=3)
```

You might want some of the fields to take default values, so you don't have to specify them each time. This can be easily done using:

```
@interface Note {
   String author() default "TCSDEVELOPER";
   String note();
   int version();
}
```

And then, if you don't specify the author, the default will be used:

```
@Note(note="Working fine", version=1) // "TCSDEVELOPER" is used
```

If your annotation just takes one parameter, you can omit its name when using it, however for this purpouse, the name of the field must be "value":

```
@interface ExpectedException {

    Class value() default String.class;

}
```

With this declaration, the annotation can be used as shown in a previous example, saving some typing:

```
@ExpectedException(NullPointerException.class)
```

Also annotations without parameters can be declared, and they're very useful to mark something, for example that a method must be tested. The annotations we've defined can be used in fields, methods, classes, etc. Often you want the annotation to be used just in some of those targets. This is done using the @Target annotation on our annotation:

```
@Target(ElementType.METHOD)

@interface Note {

   String author() default "TCSDEVELOPER";

   String note();

   int version();

}
```

Now, the Note annotation can be used just for methods. You can also specify more than one element type using the following syntax:

```
@Target({ElementType.TYPE, ElementType.METHOD})
```

Annotations can have different Retention Policies, which specify when the annotation is discarded. The SOURCE retention policy makes the annotation just available in the source code, discarting it on the compilation. This could be used in a similar way as javadocs are used: a tool that extracts information from source code to do something. The CLASS retention police, used by default, stores the annotation in the .class file but it won't be available in the JVM. The last retention policy is RUNTIME, that makes the annotation to be available also in the JVM. The retention policy is specified using the @Retention annotation.

For example:

```
@Retention(RetentionPolicy.RUNTIME)
```

We've learned to declare and use annotations; however they do nothing by themselves. Now, we need to access them to do something, and this is done using reflection. The AnnotatedElement interface represents an element that can be annoted, and thus it declares the methods we need to retrieve the annotations. The reflection classes (Class, Constructor, Field, Method, Package) implement this interface.

For example, this is a sample program that gets the Note annotations for the AnnotDemo class and shows them:

```
@Note(note="Finish this class!", version=1)

public class AnnotDemo {

    public static void main (String args[]) {

      Note note = AnnotDemo.class.getAnnotation(Note.class);
```

```
        if (note != null) {
            System.out.println("Author=" + note.author());
            System.out.println("Note=" + note.note());
            System.out.println("Version=" + note.version());
        } else {
            System.out.println("Note not found.");
        }
    }
}
```

If you run this and it says that the note was not found don't panic. Try to discover what the problem is. Did you find it? If the Note annotation doesn't have a retention policy specified, it uses CLASS by default, so the annotations are not available in the JVM. Just specify the RUNTIME retention police and it will work. The important line in the above code is:

```
Note note = AnnotDemo.class.getAnnotation(Note.class);
```

With AnnotDemo.class we get a Class object representing that class, and the getAnnotation method asks for an annotation of the type specified in the argument. That method retrieves the note if it exists or null if there are no notes of such type. The AnnotatedElement interface provides more methods in order to retrieve all the annotations for an element (without specifying the type) and to know if an annotation is present. For more information refer to the API documentation of this interface.

- Java defines seven built in annotations.
- In java.lang.annotation: @Retention, @Documented, @Target, @Inherited
- In java.lang: @Override, @Deprecated, @SuppressWarnings

Some of them were already explained on this feature. For the rest, please see the API documentation.

If you want to go deeper in this interesting subject, I recommend you to start by doing a mini-Junit class with annotations. You have to look for a @Test annotation in each method of the class, and if found, execute that method. Then, you can improve it by using the @ExpectedException as we defined above.

Static Import

Let's do some math:

```
x = Math.cos(d * Math.PI) * w + Math.sin(Math.sqrt(Math.abs(d))) * h;
```

This looks horrible! Not only is the formula hard to understand, but also the "Math." takes a lot of space and doesn't help to understand. It will be much clearer if you could do:

```
x = cos(d * PI) * w + sin (sqrt(abs(d))) * h;
```

Well, with Java 1.5 this is possible. You just have to do a static import over the members you're using, so you must add at the beginning of the file:

```
import static java.lang.Math.cos;
import static java.lang.Math.sin;
import static java.lang.Math.abs;
import static java.lang.Math.sqrt;
import static java.lang.Math.PI;
```

Those lines make the static methods and constants visible without the need of the class name. You could also do:

```
import static java.lang.Math.*;
```

But this is not recommended; almost sure you're importing a lot of things that you don't need and you don't event know that you're importing. Also, when reading the code, if you used ".*" in many static import's, you won't know from which one an attribute or method is coming. Also this could result in ambiguities when there is a static member with the same name in two different places. You should use static imports with care, just when they'll make your code look clearer because you're repeatedly accessing some static members of a class. Abusing of the import static recourse will make the code less clear.

Updates in the API

A lot of updates were done in the API, adding methods, classes and packages. The new API definition is backwards compatible, so your previous code must compile without any modifications. We'll briefly see some of the most important changes. If you want to know the full list of updates, please see http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html#base_libs

Formatted I/O

The new class java.util.Formatter enables you to use formatted strings in the old and loved C style. For example, you can do:

```
Formatter f = new Formatter ();

f.format("Decimal: %d, hexa: %x, character %c, double %1.4f.",

50,50,'A', 3.1415926);

System.out.println (f.toString());
```

The constructor of Formatter can be used in different ways so that the buffer for the formatted output is somewhere else, for example in a file. If you want to show the input on the screen, instead of the above form, you can do it directly with printf:

```
System.out.printf("Decimal: %d, hexa: %x, character %c,

double %1.4f.", 50,50,'A', 3.1415926);
```

Although this is similar to the formatting strings in C, there are some differences and improvements, so you should have a look at the Formatter class documentation for more detail. On the other side, you can read formatted input and convert to its binary form using the java.util.Scanner. For example, if you have any strings containing an unknown number of integers, you can sum them using:

```
int x=0;

Scanner scan = new Scanner("10 5 3 1");

while (scan.hasNextInt()) {

   x+=scan.nextInt();

}
```

When you create the Scanner object, you must specify a string to read from, or some other source like an InputStream. Then, you have methods to determinate if you have an element of a specified type using hasNext*, and you can read those elements with the methods next*. Of course that if you know beforehand the number of each type, you don't need the hasNext* methods. Let's see another example, where we know that we have a string and two doubles, separated either by ";" or "&":

```
Scanner scan = new Scanner("Hello World;15.4&8.4");

scan.useLocale(Locale.US);

scan.useDelimiter("[;&]");

String s;

double x,y;
```

```
s = scan.next();

x = scan.nextDouble();

y = scan.nextDouble();
```

Here, we set the locale to be US, so that the decimal separator is the dot. You have to be very careful with locales, because this might work perfect in your computer but in someone else's it might throw an exception because his locale is set to use the comma as decimal separator. Then, we set the delimiter with the method useDelimiter, that takes a regular expression pattern. As we know that we'll have a string and two doubles, we just read them from the scanner.

Collections
The Collections package has changed from its roots, by supporting generic types. The old code will compile and work, but it will produce compiler warnings. New code should be written using generics.

For example, here we create a list of Integers and then show it:

```
List<Integer> l = new ArrayList<Integer>();

l.add(10);

l.add(20);



for (int value : l) {

   System.out.println(value);

}
```

Notice that we can add ints directly thanks to autoboxing, but the type of the collection is always a class. Some other collection classes, like HashMap, take two types. See the generics section for an example. Also, new class collections are defined: enumMap and enumSet to make working with maps keys and sets in enums more efficient; AbstractQueue that provides some basic common functionality of the queues, and PriorityQueue, which can be very useful in SRMs.

Let's see an example of a PriorityQueue:

```
PriorityQueue<Double> pq = new PriorityQueue<Double>();

pq.add(4.5);

pq.add(9.3);

pq.add(1.7);



while(true) {

   Double d = pq.poll();

   if (d == null) break;

   System.out.println(d);

}
```

Adding elements it's done - not very surprisingly - with the add method. Then, the poll method retrieves the next element in the priority queue or null if it's empty, and removes it. You can also use the peek method, which works like poll but doesn't remove the element. There are also new methods in the Collections utility class. Two of them can be particularly useful in SRM's: the frequency method that counts the number of times that an element appears in a collection, and the disjoint method that returns whether two collections have no common elements. The other two added methods are addAll, which enables you to add all the elements in an array to a collection, and reverseOrder, that returns a Comparator representing the reverse order of the comparator provided as parameter.