**Chapter 18:More Language Features**

In this tutorial we will see the upgradation made in Java language since jdk 1.5 version. The new features such as Generics, Type-Safe Enumerations, Automatic Boxing and Unboxing, Annotations, For/In loops, and Static Imports.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | |  | | --- | | **0. Generics**    :    [**Link**](http://www.javatpoint.com/generics-in-java)  [**Link**](http://docs.oracle.com/javase/tutorial/java/javaOO/QandE/creating-questions.html) |   Generics in Java  The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects.  Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects.  **Advantage of Java Generics**  There are mainly 3 advantages of generics. They are as follows:  **1) Type-safety :** We can hold only a single type of objects in generics. It doesn’t allow to store other objects.  **2) Type casting is not required:** There is no need to typecast the object.  Before Generics, we need to type cast.   1. List list = **new** ArrayList(); 2. list.add("hello"); 3. String s = (String) list.get(0);//typecasting   After Generics, we don't need to typecast the object.   1. List<String> list = **new** ArrayList<String>(); 2. list.add("hello"); 3. String s = list.get(0);   **3) Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.   1. List<String> list = **new** ArrayList<String>(); 2. list.add("hello"); 3. list.add(32);//Compile Time Error   **Syntax** to use generic collection   1. ClassOrInterface<Type>   **Example** to use Generics in java   1. ArrayList<String>   Full Example of Generics in Java  Here, we are using the ArrayList class, but you can use any collection class such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, Comparator etc.   1. **import** java.util.\*; 2. **class** TestGenerics1{ 3. **public** **static** **void** main(String args[]){ 4. ArrayList<String> list=**new** ArrayList<String>(); 5. list.add("rahul"); 6. list.add("jai"); 7. //list.add(32);//compile time error 9. String s=list.get(1);//type casting is not required 10. System.out.println("element is: "+s); 12. Iterator<String> itr=list.iterator(); 13. **while**(itr.hasNext()){ 14. System.out.println(itr.next()); 15. } 16. } 17. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestGenerics1)  Output:element is: jai  rahul  jai  Example of Java Generics using Map  Now we are going to use map elements using generics. Here, we need to pass key and value. Let us understand it by a simple example:   1. **import** java.util.\*; 2. **class** TestGenerics2{ 3. **public** **static** **void** main(String args[]){ 4. Map<Integer,String> map=**new** HashMap<Integer,String>(); 5. map.put(1,"vijay"); 6. map.put(4,"umesh"); 7. map.put(2,"ankit"); 9. //Now use Map.Entry for Set and Iterator 10. Set<Map.Entry<Integer,String>> set=map.entrySet(); 12. Iterator<Map.Entry<Integer,String>> itr=set.iterator(); 13. **while**(itr.hasNext()){ 14. Map.Entry e=itr.next();//no need to typecast 15. System.out.println(e.getKey()+" "+e.getValue()); 16. } 18. }}   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestGenerics2)  Output:1 vijay  2 ankit  4 umesh  Generic class  A class that can refer to any type is known as generic class. Here, we are using **T** type parameter to create the generic class of specific type.  Let’s see the simple example to create and use the generic class.  **Creating generic class:**   1. **class** MyGen<T>{ 2. T obj; 3. **void** add(T obj){**this**.obj=obj;} 4. T get(){**return** obj;} 5. }   The T type indicates that it can refer to any type (like String, Integer, Employee etc.). The type you specify for the class, will be used to store and retrieve the data.  **Using generic class:**  Let’s see the code to use the generic class.   1. **class** TestGenerics3{ 2. **public** **static** **void** main(String args[]){ 3. MyGen<Integer> m=**new** MyGen<Integer>(); 4. m.add(2); 5. //m.add("vivek");//Compile time error 6. System.out.println(m.get()); 7. }}   Output:2  Type Parameters  The type parameters naming conventions are important to learn generics thoroughly. The commonly type parameters are as follows:   1. T - Type 2. E - Element 3. K - Key 4. N - Number 5. V - Value   Generic Method  Like generic class, we can create generic method that can accept any type of argument.  Let’s see a simple example of java generic method to print array elements. We are using here **E** to denote the element.   1. **public** **class** TestGenerics4{ 3. **public** **static** < E > **void** printArray(E[] elements) { 4. **for** ( E element : elements){ 5. System.out.println(element ); 6. } 7. System.out.println(); 8. } 9. **public** **static** **void** main( String args[] ) { 10. Integer[] intArray = { 10, 20, 30, 40, 50 }; 11. Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' }; 13. System.out.println( "Printing Integer Array" ); 14. printArray( intArray  ); 16. System.out.println( "Printing Character Array" ); 17. printArray( charArray ); 18. } 19. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestGenerics4)  Output:Printing Integer Array  10  20  30  40  50  Printing Character Array  J  A  V  A  T  P  O  I  N  T  Wildcard in Java Generics  The ? (question mark) symbol represents wildcard element. It means any type. If we write  <? extends Number>, it means any child class of Number e.g. Integer, Float, double etc. Now we can call the method of Number class through any child class object.  Let's understand it by the example given below:   1. **import** java.util.\*; 2. **abstract** **class** Shape{ 3. **abstract** **void** draw(); 4. } 5. **class** Rectangle **extends** Shape{ 6. **void** draw(){System.out.println("drawing rectangle");} 7. } 8. **class** Circle **extends** Shape{ 9. **void** draw(){System.out.println("drawing circle");} 10. }  13. **class** GenericTest{ 14. //creating a method that accepts only child class of Shape 15. **public** **static** **void** drawShapes(List<? **extends** Shape> lists){ 16. **for**(Shape s:lists){ 17. s.draw();//calling method of Shape class by child class instance 18. } 19. } 20. **public** **static** **void** main(String args[]){ 21. List<Rectangle> list1=**new** ArrayList<Rectangle>(); 22. list1.add(**new** Rectangle()); 24. List<Circle> list2=**new** ArrayList<Circle>(); 25. list2.add(**new** Circle()); 26. list2.add(**new** Circle()); 28. drawShapes(list1); 29. drawShapes(list2); 30. }}   drawing rectangle  drawing circle  drawing circle Answers to Questions and Exercises: ClassesQuestions  1. Consider the following class: 2. public class IdentifyMyParts { 3. public static int x = 7; 4. public int y = 3; 5. }    1. **Question**: What are the class variables?   **Answer**: x   * 1. **Question**: What are the instance variables?   **Answer**: y   * 1. **Question**: What is the output from the following code:   2. IdentifyMyParts a = new IdentifyMyParts();   3. IdentifyMyParts b = new IdentifyMyParts();   4. a.y = 5;   5. b.y = 6;   6. a.x = 1;   7. b.x = 2;   8. System.out.println("a.y = " + a.y);   9. System.out.println("b.y = " + b.y);   10. System.out.println("a.x = " + a.x);   11. System.out.println("b.x = " + b.x);   12. System.out.println("IdentifyMyParts.x = " + IdentifyMyParts.x);   **Answer**: Here is the output:  a.y = 5  b.y = 6  a.x = 2  b.x = 2  IdentifyMyParts.x = 2  Because x is defined as a public static int in the class IdentifyMyParts, every reference to x will have the value that was last assigned because x is a static variable (and therefore a class variable) shared across all instances of the class. That is, there is only one x: when the value of x changes in any instance it affects the value of x for all instances of IdentifyMyParts.  This is covered in the Class Variables section of [Understanding Instance and Class Members](http://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html). Exercises  1. **Exercise**: Write a class whose instances represent a single playing card from a deck of cards. Playing cards have two distinguishing properties: rank and suit. Be sure to keep your solution as you will be asked to rewrite it in [Enum Types](http://docs.oracle.com/javase/tutorial/java/javaOO/QandE/enum-questions.html).   **Answer**: [Card.java](http://docs.oracle.com/javase/tutorial/java/javaOO/examples/Card.java)(in a .java source file).   1. **Exercise**: Write a class whose instances represents a **full** deck of cards. You should also keep this solution.   **Answer**: See [Deck.java](http://docs.oracle.com/javase/tutorial/java/javaOO/examples/Deck.java)(in a .java source file).   1. **Exercise**: Write a small program to test your deck and card classes. The program can be as simple as creating a deck of cards and displaying its cards.   **Answer**: See [DisplayDeck.java](http://docs.oracle.com/javase/tutorial/java/javaOO/examples/DisplayDeck.java)(in a .java source file). | | |  | | --- | | **0. For-each loop**    :    [**Link**](http://www.tutorialspoint.com/javaexamples/method_for.htm) |  Problem Description: How to use for and foreach loops to display elements of an array. Solution: This example displays an integer array using for loop & foreach loops.  public class Main {  public static void main(String[] args) {  int[] intary = { 1,2,3,4};  forDisplay(intary);  foreachDisplay(intary);  }  public static void forDisplay(int[] a){  System.out.println("Display an array using for loop");  for (int i = 0; i < a.length; i++) {  System.out.print(a[i] + " ");  }  System.out.println();  }  public static void foreachDisplay(int[] data){  System.out.println("Display an array using for  each loop");  for (int a : data) {  System.out.print(a+ " ");  }  }  } Result: The above code sample will produce the following result.  Display an array using for loop  1 2 3 4  Display an array using for each loop  1 2 3 4 | | |  | | --- | | **0. Auto boxing/unboxing**    :    [**Link**](http://www.cs.indiana.edu/classes/jett/sstamm/#_boxing) |  1.3 Autoboxing/Unboxing Integer i = new Integer(4);  int j = i.intValue();  Number n = new Float(3.14159);  Boolean stuff = new Boolean(false);  // stuff before ? must be a boolean (lower case)  System.out.println( stuff.booleanValue() ? "Yep" : "Nope" );  Sick of this? Me too. Do this instead:  Integer i = 4;  int j = i;  Number n = 3.14159f;  Boolean stuff = false;  System.out.println( stuff ? "Yep" : "Nope" );  This is pretty nice. Especially since you can use ++ and other similar operators with the wrapper types now too. 1.4 Typesafe Enums Enums are just magic classes to help prevent the methodless-interface antipattern. They let you make classes that will enumerate values, but also keep the types specific. Before, we could simulate enums with a bunch of static final int variables or something. The problem with those is that you could confuse **any** int with one of the constants. With enumerations, only the values in the enum are valid. For example:  public enum JettStaff {  ADRIAN,  ARIJIT,  BETH,  ERIC,  KATIE,  KATY,  RAJA,  RICH,  SUZANNE  };  JettStaff x = JettStaff.SUZANNE;  Now, it gets even cooler. I don't have to keep track of separate information to store, say the peoples' full names. I can associate them directly, just like in a class! Each of the values of JettStaffare instances of the JettStaff enumeration, so we can define a constructor and a toString() method.  public enum JettStaff {  ADRIAN("Adrian German"),  ARIJIT("Arijit Sengupta"),  BETH("Beth Plale"),  ERIC("Eric Wernert"),  KATIE("Katie A. Siek"),  KATY("Katy Borner"),  RAJA("Raja Sooriamurthi"),  RICH("Rich Kick"),  SUZANNE("Suzanne Menzel");  private String name;    public JettStaff(String n) { this.name = n; }  public String toString() { return this.name; }  }  JettStaff x = JettStaff.SUZANNE;  System.out.println(x);  *But wait, it gets cooler!* Now you can also give each enumerated value a custom body. Since they're each instances, you could design a toString() method for each:  public enum JettStaff {  ADRIAN("Adrian German") {  public String toString() {  return name + " (dgerman@indiana.edu)";  }  },  ARJIT("Arjit Sengupta") {  public String toString() {  return name + " (asengupt@indiana.edu)";  }  },    // and on for the rest...  private String name;    public JettStaff(String n) { this.name = n; }  }  JettStaff x = JettStaff.SUZANNE;  System.out.println(x);  Last but not least, enums can extend each other. Imagine that! 1.5 Varargs What is your impression of "..."? It's a nice little note that might come in handy. Notice how when you pass arguments from the command line ...  C:/> java MyProg a b c  You gave me 3 args! Yay.  ...  ... you don't have to pass an array of stuff. The runtime automatically converts the arguments into an array of strings. You can do that now in all of your methods! For example, instead of doing this:  public class VarArgs {  public static void main(String[] args) {  String[] newArgs = {"a", "b", "c"};  vaMethod(newArgs);  }  public void vaMethod(String[] args) {  System.out.println("You gave me " + args.length + " args! Yay.");  }  }  You can declare it more easily, and not have to construct the array ahead of time:  public class VarArgs {  public static void main(String[] args) {  vaMethod("a", "b", "c");  }  public void vaMethod(String... args) {  System.out.println("You gave me " + args.length + " args! Yay.");  }  }  Notice that when you use the ... syntax, it automatically treats that parameter as an array but you don't have to pass it in that way. Nifty. Let's add one of those for/in loops:  public class VarArgs {  public static void main(String[] args) {  vaMethod("a", "b", "c");  }  public void vaMethod(String... args) {  for(String s : args)  System.out.println(s);  }  } 1.6 Static Import Remember making all those interfaces that just have constants in them?  import java.awt.\*;  public interface BouncingBallConstants {  public static final Color BACK\_COLOR = Color.WHITE;  public static final Color BALL\_COLOR = Color.BLUE;  public static final int BALL\_SIZE = 50;  }  Scrap that. Put these into a **REAL** class and then just import the static members from all the other ones using import static <Package or Class>;. This addition is pretty straightforward, and it helps prevent bloat and the "methodless interface" design antipattern. 1.7 Annotations (Metadata) Now for the weirdest part. Annotations are not really something that will affect how you program in Java, unless you need to associate some sort of *metadata* or annotations with classes, methods, variables, etc.  So what are annotations anyway? That's a good question. They provide a little extra information about the classes you write, and a class can use the Reflection package later to read the annotations. These are useful because you can attach extra information to your code that may determine how it is used or maybe if it is used at all.  For example, in J2SE 5, you can *declare* your intent to override a method like toString() in one of your classes:  public class MyClass extends Object {  @Override  public String toString() {  return "My overridden method!";  }  }  In the above example, we declare that we will override the immediately following toString() method. So the compiler looks in our superclass (Object) for the same metho and makes sure it exists. If for some reason we had *overloaded* toString() by declaring it with different parameters and maybe return type, then the compiler would throw an error saying we didn't override correctly. This is really useful if you want to make sure you override a method as opposed to simply over*loading* it.  Of course you can define your own annotations. They're basically like interfaces, but they can contain values. An example annotation looks like:  public @interface Conference {  String what();  String when();  String location();  }  This annotation declares three members: what, when, location and sets them up to have "getters" and "setters" automatically! That means each @Conference annotation has those three fields associated with it, and I don't have to define the accessor and mutator methods to set them up (see the next code listing). If I define this annotation like this, I can use it to mark code that I use for the Jett conference:  @Conference(what="JETT",  when="November 2004",  location="IUB")  public class MyMagicJettClass {  //...  }  And now the @Conference type of data is associated with my class. Later on, I could write an analyzer that goes through all of my code and lets me know which classes were used at conferences as well as which conferences they were used at and when. This specific example doesn't have any effect on the way MyMagicJettClass operates.  So the annotations require two-fold cooperation: the programmer must properly annotate her code to provide adequate metadata needed and other developers who will want to know this metadata must know how to extract it using the Java Reflection package. That's a whole hours-long session on how to extract them, so I'm not going to go into depth here.  Where are they useful? Say you are working with RMI (Remote Method Invocation) and you don't want all of your methods available remotely. You could annotate the remotable ones with a@Remote annotation, then whatever serves up the remote access can only allow those to be remotely accessed. There are a ton of great uses for these, and they are fully extendable (you can annotate annotations)! 2. How it will affect you NOW All of the previous Java language features will still work. Some are deprecated, but not very many. You should also know that the 2004 exam was written before 1.5 was available, so its features are not at all critical.  The topics covered by the exam don't include generic typing, annotations/metadata or variable arguments (since really they're just language features and not so much programming constructs). This is good news, since if your students don't immediately master these new features it won't harm them.  What if my students use new 1.5 language features on the exam? If they're used properly (and the code is well-designed), the readers will most likely not deduct any points. But in the end the deductions are up to the readers. Automatic boxing and unboxing as well as correct use of the for/in loop will be acceptable.  The biggest problem would be a student who tries to use one of these new features (such as generics or enums) but misuses them terribly, obscuring the code. Then a reader might not at all understand what constructs the student was trying to use and mistake it for non-code garbage. Because the readers are as new to the 1.5 features as we are, we cannot expect them to have mastered everything to the point of being able to identify misuse of one. 2.1. To add to your lesson plan... You may not want to add much of this to your lesson plan immediately, since most of it is not necessary for good programming. Sneaking in the for/in loop and autoboxing and unboxing would be quite simple (since the students are likely to leave out the manual boxing and unboxing regardless if the compiler doesn't complain). At least for the time being, if the students can understand Java and write good 1.4 code, it will most likely be acceptable. 3. How it will affect you eventually... Nobody knows. Really it's up to the AP board and college CS departments where we go from here. How much of this new Java stuff becomes mainstream and intravenous so that we breathe it like objects? It is my personal opinion that the generics, for/in loop, autoboxing static import and varargs stuff is all syntax sugar: it's just a little bit of convienience and not a really big construct such as other big paradigm shifts like procedural to object oriented programming. 3.1. How can the new features help? So now I step off my soapbox and ask you to think of how these new features might streamline your teaching methods. How can you work these into your lesson plan to help teach the ideas of polymorphism, generic types, AOP (Aspect oriented programming: the annotations stuff), and enumerations? In what ways can they help you teach the current topics in a more streamlined way? | | |  | | --- | | **0. Enumerated types**    :    [**Link**](http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html) |   **Enum Types**  An *enum type* is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it. Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week.  Because they are constants, the names of an enum type's fields are in uppercase letters.  In the Java programming language, you define an enum type by using the enum keyword. For example, you would specify a days-of-the-week enum type as:  public enum Day {  SUNDAY, MONDAY, TUESDAY, WEDNESDAY,  THURSDAY, FRIDAY, SATURDAY  }  You should use enum types any time you need to represent a fixed set of constants. That includes natural enum types such as the planets in our solar system and data sets where you know all possible values at compile time—for example, the choices on a menu, command line flags, and so on.  Here is some code that shows you how to use the Day enum defined above:  public class EnumTest {  Day day;    public EnumTest(Day day) {  this.day = day;  }    public void tellItLikeItIs() {  switch (day) {  case MONDAY:  System.out.println("Mondays are bad.");  break;    case FRIDAY:  System.out.println("Fridays are better.");  break;    case SATURDAY: case SUNDAY:  System.out.println("Weekends are best.");  break;    default:  System.out.println("Midweek days are so-so.");  break;  }  }    public static void main(String[] args) {  EnumTest firstDay = new EnumTest(Day.MONDAY);  firstDay.tellItLikeItIs();  EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);  thirdDay.tellItLikeItIs();  EnumTest fifthDay = new EnumTest(Day.FRIDAY);  fifthDay.tellItLikeItIs();  EnumTest sixthDay = new EnumTest(Day.SATURDAY);  sixthDay.tellItLikeItIs();  EnumTest seventhDay = new EnumTest(Day.SUNDAY);  seventhDay.tellItLikeItIs();  }  }  The output is:  Mondays are bad.  Midweek days are so-so.  Fridays are better.  Weekends are best.  Weekends are best.  Java programming language enum types are much more powerful than their counterparts in other languages. The enum declaration defines a *class* (called an *enum type*). The enum class body can include methods and other fields. The compiler automatically adds some special methods when it creates an enum. For example, they have a static valuesmethod that returns an array containing all of the values of the enum in the order they are declared. This method is commonly used in combination with the for-each construct to iterate over the values of an enum type. For example, this code from the Planet class example below iterates over all the planets in the solar system.  for (Planet p : Planet.values()) {  System.out.printf("Your weight on %s is %f%n",  p, p.surfaceWeight(mass));  }  **Note:** *All* enums implicitly extend java.lang.Enum. Because a class can only extend one parent (see [Declaring Classes](http://docs.oracle.com/javase/tutorial/java/javaOO/classdecl.html)), the Java language does not support multiple inheritance of state (see [Multiple Inheritance of State, Implementation, and Type](http://docs.oracle.com/javase/tutorial/java/IandI/multipleinheritance.html)), and therefore an enum cannot extend anything else.  In the following example, Planet is an enum type that represents the planets in the solar system. They are defined with constant mass and radius properties.  Each enum constant is declared with values for the mass and radius parameters. These values are passed to the constructor when the constant is created. Java requires that the constants be defined first, prior to any fields or methods. Also, when there are fields and methods, the list of enum constants must end with a semicolon.  **Note:** The constructor for an enum type must be package-private or private access. It automatically creates the constants that are defined at the beginning of the enum body. You cannot invoke an enum constructor yourself.  In addition to its properties and constructor, Planet has methods that allow you to retrieve the surface gravity and weight of an object on each planet. Here is a sample program that takes your weight on earth (in any unit) and calculates and prints your weight on all of the planets (in the same unit):  public enum Planet {  MERCURY (3.303e+23, 2.4397e6),  VENUS (4.869e+24, 6.0518e6),  EARTH (5.976e+24, 6.37814e6),  MARS (6.421e+23, 3.3972e6),  JUPITER (1.9e+27, 7.1492e7),  SATURN (5.688e+26, 6.0268e7),  URANUS (8.686e+25, 2.5559e7),  NEPTUNE (1.024e+26, 2.4746e7);  private final double mass; // in kilograms  private final double radius; // in meters  Planet(double mass, double radius) {  this.mass = mass;  this.radius = radius;  }  private double mass() { return mass; }  private double radius() { return radius; }  // universal gravitational constant (m3 kg-1 s-2)  public static final double G = 6.67300E-11;  double surfaceGravity() {  return G \* mass / (radius \* radius);  }  double surfaceWeight(double otherMass) {  return otherMass \* surfaceGravity();  }  public static void main(String[] args) {  if (args.length != 1) {  System.err.println("Usage: java Planet <earth\_weight>");  System.exit(-1);  }  double earthWeight = Double.parseDouble(args[0]);  double mass = earthWeight/EARTH.surfaceGravity();  for (Planet p : Planet.values())  System.out.printf("Your weight on %s is %f%n",  p, p.surfaceWeight(mass));  }  }  If you run Planet.class from the command line with an argument of 175, you get this output:  $ java Planet 175  Your weight on MERCURY is 66.107583  Your weight on VENUS is 158.374842  Your weight on EARTH is 175.000000  Your weight on MARS is 66.279007  Your weight on JUPITER is 442.847567  Your weight on SATURN is 186.552719  Your weight on URANUS is 158.397260  Your weight on NEPTUNE is 199.207413 | | |  | | --- | | **0. Varargs**    :    [**Link**](http://www.cs.indiana.edu/classes/jett/sstamm/#_varargs) |  1.5 Varargs What is your impression of "..."? It's a nice little note that might come in handy. Notice how when you pass arguments from the command line ...  C:/> java MyProg a b c  You gave me 3 args! Yay.  ...  ... you don't have to pass an array of stuff. The runtime automatically converts the arguments into an array of strings. You can do that now in all of your methods! For example, instead of doing this:  public class VarArgs {  public static void main(String[] args) {  String[] newArgs = {"a", "b", "c"};  vaMethod(newArgs);  }  public void vaMethod(String[] args) {  System.out.println("You gave me " + args.length + " args! Yay.");  }  }  You can declare it more easily, and not have to construct the array ahead of time:  public class VarArgs {  public static void main(String[] args) {  vaMethod("a", "b", "c");  }  public void vaMethod(String... args) {  System.out.println("You gave me " + args.length + " args! Yay.");  }  }  Notice that when you use the ... syntax, it automatically treats that parameter as an array but you don't have to pass it in that way. Nifty. Let's add one of those for/in loops:  public class VarArgs {  public static void main(String[] args) {  vaMethod("a", "b", "c");  }  public void vaMethod(String... args) {  for(String s : args)  System.out.println(s);  }  } 1.6 Static Import Remember making all those interfaces that just have constants in them?  import java.awt.\*;  public interface BouncingBallConstants {  public static final Color BACK\_COLOR = Color.WHITE;  public static final Color BALL\_COLOR = Color.BLUE;  public static final int BALL\_SIZE = 50;  }  Scrap that. Put these into a **REAL** class and then just import the static members from all the other ones using import static <Package or Class>;. This addition is pretty straightforward, and it helps prevent bloat and the "methodless interface" design antipattern. 1.7 Annotations (Metadata) Now for the weirdest part. Annotations are not really something that will affect how you program in Java, unless you need to associate some sort of *metadata* or annotations with classes, methods, variables, etc.  So what are annotations anyway? That's a good question. They provide a little extra information about the classes you write, and a class can use the Reflection package later to read the annotations. These are useful because you can attach extra information to your code that may determine how it is used or maybe if it is used at all.  For example, in J2SE 5, you can *declare* your intent to override a method like toString() in one of your classes:  public class MyClass extends Object {  @Override  public String toString() {  return "My overridden method!";  }  }  In the above example, we declare that we will override the immediately following toString() method. So the compiler looks in our superclass (Object) for the same metho and makes sure it exists. If for some reason we had *overloaded* toString() by declaring it with different parameters and maybe return type, then the compiler would throw an error saying we didn't override correctly. This is really useful if you want to make sure you override a method as opposed to simply over*loading* it.  Of course you can define your own annotations. They're basically like interfaces, but they can contain values. An example annotation looks like:  public @interface Conference {  String what();  String when();  String location();  }  This annotation declares three members: what, when, location and sets them up to have "getters" and "setters" automatically! That means each @Conference annotation has those three fields associated with it, and I don't have to define the accessor and mutator methods to set them up (see the next code listing). If I define this annotation like this, I can use it to mark code that I use for the Jett conference:  @Conference(what="JETT",  when="November 2004",  location="IUB")  public class MyMagicJettClass {  //...  }  And now the @Conference type of data is associated with my class. Later on, I could write an analyzer that goes through all of my code and lets me know which classes were used at conferences as well as which conferences they were used at and when. This specific example doesn't have any effect on the way MyMagicJettClass operates.  So the annotations require two-fold cooperation: the programmer must properly annotate her code to provide adequate metadata needed and other developers who will want to know this metadata must know how to extract it using the Java Reflection package. That's a whole hours-long session on how to extract them, so I'm not going to go into depth here.  Where are they useful? Say you are working with RMI (Remote Method Invocation) and you don't want all of your methods available remotely. You could annotate the remotable ones with a@Remote annotation, then whatever serves up the remote access can only allow those to be remotely accessed. There are a ton of great uses for these, and they are fully extendable (you can annotate annotations)! 2. How it will affect you NOW All of the previous Java language features will still work. Some are deprecated, but not very many. You should also know that the 2004 exam was written before 1.5 was available, so its features are not at all critical.  The topics covered by the exam don't include generic typing, annotations/metadata or variable arguments (since really they're just language features and not so much programming constructs). This is good news, since if your students don't immediately master these new features it won't harm them.  What if my students use new 1.5 language features on the exam? If they're used properly (and the code is well-designed), the readers will most likely not deduct any points. But in the end the deductions are up to the readers. Automatic boxing and unboxing as well as correct use of the for/in loop will be acceptable.  The biggest problem would be a student who tries to use one of these new features (such as generics or enums) but misuses them terribly, obscuring the code. Then a reader might not at all understand what constructs the student was trying to use and mistake it for non-code garbage. Because the readers are as new to the 1.5 features as we are, we cannot expect them to have mastered everything to the point of being able to identify misuse of one. 2.1. To add to your lesson plan... You may not want to add much of this to your lesson plan immediately, since most of it is not necessary for good programming. Sneaking in the for/in loop and autoboxing and unboxing would be quite simple (since the students are likely to leave out the manual boxing and unboxing regardless if the compiler doesn't complain). At least for the time being, if the students can understand Java and write good 1.4 code, it will most likely be acceptable. 3. How it will affect you eventually... Nobody knows. Really it's up to the AP board and college CS departments where we go from here. How much of this new Java stuff becomes mainstream and intravenous so that we breathe it like objects? It is my personal opinion that the generics, for/in loop, autoboxing static import and varargs stuff is all syntax sugar: it's just a little bit of convienience and not a really big construct such as other big paradigm shifts like procedural to object oriented programming. 3.1. How can the new features help? So now I step off my soapbox and ask you to think of how these new features might streamline your teaching methods. How can you work these into your lesson plan to help teach the ideas of polymorphism, generic types, AOP (Aspect oriented programming: the annotations stuff), and enumerations? In what ways can they help you teach the current topics in a more streamlined way? | | |  | | --- | | **0. Static imports**    :    [**Link**](http://www.cs.indiana.edu/classes/jett/sstamm/#_staticimport) |  1.6 Static Import Remember making all those interfaces that just have constants in them?  import java.awt.\*;  public interface BouncingBallConstants {  public static final Color BACK\_COLOR = Color.WHITE;  public static final Color BALL\_COLOR = Color.BLUE;  public static final int BALL\_SIZE = 50;  }  Scrap that. Put these into a **REAL** class and then just import the static members from all the other ones using import static <Package or Class>;. This addition is pretty straightforward, and it helps prevent bloat and the "methodless interface" design antipattern. 1.7 Annotations (Metadata) Now for the weirdest part. Annotations are not really something that will affect how you program in Java, unless you need to associate some sort of *metadata* or annotations with classes, methods, variables, etc.  So what are annotations anyway? That's a good question. They provide a little extra information about the classes you write, and a class can use the Reflection package later to read the annotations. These are useful because you can attach extra information to your code that may determine how it is used or maybe if it is used at all.  For example, in J2SE 5, you can *declare* your intent to override a method like toString() in one of your classes:  public class MyClass extends Object {  @Override  public String toString() {  return "My overridden method!";  }  }  In the above example, we declare that we will override the immediately following toString() method. So the compiler looks in our superclass (Object) for the same metho and makes sure it exists. If for some reason we had *overloaded* toString() by declaring it with different parameters and maybe return type, then the compiler would throw an error saying we didn't override correctly. This is really useful if you want to make sure you override a method as opposed to simply over*loading* it.  Of course you can define your own annotations. They're basically like interfaces, but they can contain values. An example annotation looks like:  public @interface Conference {  String what();  String when();  String location();  }  This annotation declares three members: what, when, location and sets them up to have "getters" and "setters" automatically! That means each @Conference annotation has those three fields associated with it, and I don't have to define the accessor and mutator methods to set them up (see the next code listing). If I define this annotation like this, I can use it to mark code that I use for the Jett conference:  @Conference(what="JETT",  when="November 2004",  location="IUB")  public class MyMagicJettClass {  //...  }  And now the @Conference type of data is associated with my class. Later on, I could write an analyzer that goes through all of my code and lets me know which classes were used at conferences as well as which conferences they were used at and when. This specific example doesn't have any effect on the way MyMagicJettClass operates.  So the annotations require two-fold cooperation: the programmer must properly annotate her code to provide adequate metadata needed and other developers who will want to know this metadata must know how to extract it using the Java Reflection package. That's a whole hours-long session on how to extract them, so I'm not going to go into depth here.  Where are they useful? Say you are working with RMI (Remote Method Invocation) and you don't want all of your methods available remotely. You could annotate the remotable ones with a@Remote annotation, then whatever serves up the remote access can only allow those to be remotely accessed. There are a ton of great uses for these, and they are fully extendable (you can annotate annotations)! 2. How it will affect you NOW All of the previous Java language features will still work. Some are deprecated, but not very many. You should also know that the 2004 exam was written before 1.5 was available, so its features are not at all critical.  The topics covered by the exam don't include generic typing, annotations/metadata or variable arguments (since really they're just language features and not so much programming constructs). This is good news, since if your students don't immediately master these new features it won't harm them.  What if my students use new 1.5 language features on the exam? If they're used properly (and the code is well-designed), the readers will most likely not deduct any points. But in the end the deductions are up to the readers. Automatic boxing and unboxing as well as correct use of the for/in loop will be acceptable.  The biggest problem would be a student who tries to use one of these new features (such as generics or enums) but misuses them terribly, obscuring the code. Then a reader might not at all understand what constructs the student was trying to use and mistake it for non-code garbage. Because the readers are as new to the 1.5 features as we are, we cannot expect them to have mastered everything to the point of being able to identify misuse of one. 2.1. To add to your lesson plan... You may not want to add much of this to your lesson plan immediately, since most of it is not necessary for good programming. Sneaking in the for/in loop and autoboxing and unboxing would be quite simple (since the students are likely to leave out the manual boxing and unboxing regardless if the compiler doesn't complain). At least for the time being, if the students can understand Java and write good 1.4 code, it will most likely be acceptable. 3. How it will affect you eventually... Nobody knows. Really it's up to the AP board and college CS departments where we go from here. How much of this new Java stuff becomes mainstream and intravenous so that we breathe it like objects? It is my personal opinion that the generics, for/in loop, autoboxing static import and varargs stuff is all syntax sugar: it's just a little bit of convienience and not a really big construct such as other big paradigm shifts like procedural to object oriented programming. 3.1. How can the new features help? So now I step off my soapbox and ask you to think of how these new features might streamline your teaching methods. How can you work these into your lesson plan to help teach the ideas of polymorphism, generic types, AOP (Aspect oriented programming: the annotations stuff), and enumerations? In what ways can they help you teach the current topics in a more streamlined way? | | |  | | --- | | **0. Metadata / annotations**    :    [**Link**](http://www.cs.indiana.edu/classes/jett/sstamm/#_annotations) |  1.7 Annotations (Metadata) Now for the weirdest part. Annotations are not really something that will affect how you program in Java, unless you need to associate some sort of *metadata* or annotations with classes, methods, variables, etc.  So what are annotations anyway? That's a good question. They provide a little extra information about the classes you write, and a class can use the Reflection package later to read the annotations. These are useful because you can attach extra information to your code that may determine how it is used or maybe if it is used at all.  For example, in J2SE 5, you can *declare* your intent to override a method like toString() in one of your classes:  public class MyClass extends Object {  @Override  public String toString() {  return "My overridden method!";  }  }  In the above example, we declare that we will override the immediately following toString() method. So the compiler looks in our superclass (Object) for the same metho and makes sure it exists. If for some reason we had *overloaded* toString() by declaring it with different parameters and maybe return type, then the compiler would throw an error saying we didn't override correctly. This is really useful if you want to make sure you override a method as opposed to simply over*loading* it.  Of course you can define your own annotations. They're basically like interfaces, but they can contain values. An example annotation looks like:  public @interface Conference {  String what();  String when();  String location();  }  This annotation declares three members: what, when, location and sets them up to have "getters" and "setters" automatically! That means each @Conference annotation has those three fields associated with it, and I don't have to define the accessor and mutator methods to set them up (see the next code listing). If I define this annotation like this, I can use it to mark code that I use for the Jett conference:  @Conference(what="JETT",  when="November 2004",  location="IUB")  public class MyMagicJettClass {  //...  }  And now the @Conference type of data is associated with my class. Later on, I could write an analyzer that goes through all of my code and lets me know which classes were used at conferences as well as which conferences they were used at and when. This specific example doesn't have any effect on the way MyMagicJettClass operates.  So the annotations require two-fold cooperation: the programmer must properly annotate her code to provide adequate metadata needed and other developers who will want to know this metadata must know how to extract it using the Java Reflection package. That's a whole hours-long session on how to extract them, so I'm not going to go into depth here.  Where are they useful? Say you are working with RMI (Remote Method Invocation) and you don't want all of your methods available remotely. You could annotate the remotable ones with a@Remote annotation, then whatever serves up the remote access can only allow those to be remotely accessed. There are a ton of great uses for these, and they are fully extendable (you can annotate annotations)! 2. How it will affect you NOW All of the previous Java language features will still work. Some are deprecated, but not very many. You should also know that the 2004 exam was written before 1.5 was available, so its features are not at all critical.  The topics covered by the exam don't include generic typing, annotations/metadata or variable arguments (since really they're just language features and not so much programming constructs). This is good news, since if your students don't immediately master these new features it won't harm them.  What if my students use new 1.5 language features on the exam? If they're used properly (and the code is well-designed), the readers will most likely not deduct any points. But in the end the deductions are up to the readers. Automatic boxing and unboxing as well as correct use of the for/in loop will be acceptable.  The biggest problem would be a student who tries to use one of these new features (such as generics or enums) but misuses them terribly, obscuring the code. Then a reader might not at all understand what constructs the student was trying to use and mistake it for non-code garbage. Because the readers are as new to the 1.5 features as we are, we cannot expect them to have mastered everything to the point of being able to identify misuse of one. 2.1. To add to your lesson plan... You may not want to add much of this to your lesson plan immediately, since most of it is not necessary for good programming. Sneaking in the for/in loop and autoboxing and unboxing would be quite simple (since the students are likely to leave out the manual boxing and unboxing regardless if the compiler doesn't complain). At least for the time being, if the students can understand Java and write good 1.4 code, it will most likely be acceptable. 3. How it will affect you eventually... Nobody knows. Really it's up to the AP board and college CS departments where we go from here. How much of this new Java stuff becomes mainstream and intravenous so that we breathe it like objects? It is my personal opinion that the generics, for/in loop, autoboxing static import and varargs stuff is all syntax sugar: it's just a little bit of convienience and not a really big construct such as other big paradigm shifts like procedural to object oriented programming. 3.1. How can the new features help? So now I step off my soapbox and ask you to think of how these new features might streamline your teaching methods. How can you work these into your lesson plan to help teach the ideas of polymorphism, generic types, AOP (Aspect oriented programming: the annotations stuff), and enumerations? In what ways can they help you teach the current topics in a more streamlined way? | | |  | | --- | | **0. Formatted IO**    :    [**Link**](http://docs.oracle.com/javase/tutorial/essential/io/formatting.html) | |   **Formatting**  Stream objects that implement formatting are instances of either [PrintWriter](https://docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html), a character stream class, or [PrintStream](https://docs.oracle.com/javase/8/docs/api/java/io/PrintStream.html), a byte stream class.  **Note:** The only PrintStream objects you are likely to need are [System.out](https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#out) and [System.err](https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#err). (See [I/O from the Command Line](http://docs.oracle.com/javase/tutorial/essential/io/cl.html) for more on these objects.) When you need to create a formatted output stream, instantiate PrintWriter, not PrintStream.  Like all byte and character stream objects, instances of PrintStream and PrintWriter implement a standard set of write methods for simple byte and character output. In addition, both PrintStream and PrintWriter implement the same set of methods for converting internal data into formatted output. Two levels of formatting are provided:   * print and println format individual values in a standard way. * format formats almost any number of values based on a format string, with many options for precise formatting.   **The print and println Methods**  Invoking print or println outputs a single value after converting the value using the appropriate toString method. We can see this in the [Root](http://docs.oracle.com/javase/tutorial/essential/io/examples/Root.java) example:  public class Root {  public static void main(String[] args) {  int i = 2;  double r = Math.sqrt(i);    System.out.print("The square root of ");  System.out.print(i);  System.out.print(" is ");  System.out.print(r);  System.out.println(".");  i = 5;  r = Math.sqrt(i);  System.out.println("The square root of " + i + " is " + r + ".");  }  }  Here is the output of Root:  The square root of 2 is 1.4142135623730951.  The square root of 5 is 2.23606797749979.  The i and r variables are formatted twice: the first time using code in an overload of print, the second time by conversion code automatically generated by the Java compiler, which also utilizes toString. You can format any value this way, but you don't have much control over the results.  **The format Method**  The format method formats multiple arguments based on a *format string*. The format string consists of static text embedded with *format specifiers*; except for the format specifiers, the format string is output unchanged.  Format strings support many features. In this tutorial, we'll just cover some basics. For a complete description, see [format string syntax](https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html#syntax) in the API specification.  The [Root2](http://docs.oracle.com/javase/tutorial/essential/io/examples/Root2.java) example formats two values with a single format invocation:  public class Root2 {  public static void main(String[] args) {  int i = 2;  double r = Math.sqrt(i);    System.out.format("The square root of %d is %f.%n", i, r);  }  }  Here is the output:  The square root of 2 is 1.414214.  Like the three used in this example, all format specifiers begin with a % and end with a 1- or 2-character *conversion* that specifies the kind of formatted output being generated. The three conversions used here are:   * d formats an integer value as a decimal value. * f formats a floating point value as a decimal value. * n outputs a platform-specific line terminator.   Here are some other conversions:   * x formats an integer as a hexadecimal value. * s formats any value as a string. * tB formats an integer as a locale-specific month name.   There are many other conversions.  **Note:**  Except for %% and %n, all format specifiers must match an argument. If they don't, an exception is thrown.  In the Java programming language, the \n escape always generates the linefeed character (\u000A). Don't use \n unless you specifically want a linefeed character. To get the correct line separator for the local platform, use %n.  In addition to the conversion, a format specifier can contain several additional elements that further customize the formatted output. Here's an example, [Format](http://docs.oracle.com/javase/tutorial/essential/io/examples/Format.java), that uses every possible kind of element.  public class Format {  public static void main(String[] args) {  System.out.format("%f, %1$+020.10f %n", Math.PI);  }  }  Here's the output:  3.141593, +00000003.1415926536  The additional elements are all optional. The following figure shows how the longer specifier breaks down into elements.  Elements of a format specifier  Elements of a Format Specifier.  The elements must appear in the order shown. Working from the right, the optional elements are:   * **Precision**. For floating point values, this is the mathematical precision of the formatted value. For s and other general conversions, this is the maximum width of the formatted value; the value is right-truncated if necessary. * **Width**. The minimum width of the formatted value; the value is padded if necessary. By default the value is left-padded with blanks. * **Flags** specify additional formatting options. In the Format example, the + flag specifies that the number should always be formatted with a sign, and the 0 flag specifies that 0 is the padding character. Other flags include - (pad on the right) and , (format number with locale-specific thousands separators). Note that some flags cannot be used with certain other flags or with certain conversions. * The **Argument Index** allows you to explicitly match a designated argument. You can also specify < to match the same argument as the previous specifier. Thus the example could have said: System.out.format("%f, %<+020.10f %n", Math.PI);   [« Previous](http://docs.oracle.com/javase/tutorial/essential/io/scanning.html) • [Trail](http://docs.oracle.com/javase/tutorial/essential/TOC.html) • [Next »](http://docs.oracle.com/javase/tutorial/essential/io/cl.html)  **Chapter 19:Inner Classes and Interfaces**  In this tutorial we will see what are inner classes. The syntax for writing an inner class. Different types of inner classes and their usage. Also the advantages and disadvantages of writing inner classes.   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  | | --- | | **0. Usage of Inner Classes**    :    [**Link**](http://www.javatpoint.com/java-inner-class) |  Java Inner Class  1. [Java Inner classes](http://www.javatpoint.com/java-inner-class) 2. [Advantage of Inner class](http://www.javatpoint.com/java-inner-class#nestedad) 3. [Difference between nested class and inner class](http://www.javatpoint.com/java-inner-class#nesteddiff) 4. [Types of Nested classes](http://www.javatpoint.com/java-inner-class#nestedtypes)   **Java inner class** or nested class is a class i.e. declared inside the class or interface.  We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.  Additionally, it can access all the members of outer class including private data members and methods. Syntax of Inner class  1. **class** Java\_Outer\_class{ 2. //code 3. **class** Java\_Inner\_class{ 4. //code 5. } 6. }  Advantage of java inner classes There are basically three advantages of inner classes in java. They are as follows:  1) Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.  2) Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.  3) **Code Optimization**: It requires less code to write.  Do You Know   * What is the internal code generated by the compiler for member inner class ? * What are the two ways to create annonymous inner class ? * Can we access the non-final local variable inside the local inner class ? * How to access the static nested class ? * Can we define an interface within the class ? * Can we define a class within the interface ?  Difference between nested class and inner class in Java Inner class is a part of nested class. Non-static nested classes are known as inner classes. Types of Nested classes There are two types of nested classes non-static and static nested classes.The non-static nested classes are also known as inner classes.   1. Non-static nested class(inner class)    * a)Member inner class    * b)Annomynous inner class    * c)Local inner class 2. Static nested class  |  |  | | --- | --- | | **Type** | **Description** | | [Member Inner Class](http://www.javatpoint.com/member-inner-class) | A class created within class and outside method. | | [Anonymous Inner Class](http://www.javatpoint.com/anonymous-inner-class) | A class created for implementing interface or extending  class. Its name is decided by the java compiler. | | [Local Inner Class](http://www.javatpoint.com/local-inner-class) | A class created within method. | | [Static Nested Class](http://www.javatpoint.com/static-nested-class) | A static class created within class. | | [Nested Interface](http://www.javatpoint.com/nested-interface) | An interface created within class or interface. | | | |  | | --- | | **0. Static Member Classes & Interfaces**    :    [**Link**](http://javabeginnerstutorial.com/core-java-tutorial/inner-class/)  [**Link**](http://etutorials.org/Misc/programmers+guide+java+certification/Chapter+7.+Nested+Classes+And+Interfaces/Programming+Exercise/) |   Inner classes are class within Class. Inner class instance has special relationship with Outer class. This special relationship gives inner class access to member of outer class as if they are the part of outer class. **Note: Inner class instance has access to all member of the outer class(Public, Private & Protected)**  Syntax for creating Inner Class  |  |  | | --- | --- | | 1  2  3  4  5  6 | //outer class  class OuterClass {  //inner class  class InnerClass {  }  } |    Type of Inner class  * Static * Method Local * Anonymous * Other then above these normal inner class    Normal Inner Class Inner classes which are not method local , static or anonymous are normal inner class.   |  |  | | --- | --- | | 1  2  3  4  5  6 | //outer class  class OuterClass {  //inner class  class InnerClass {  }  } |   If you compile above code it will produce two class file.   |  |  | | --- | --- | | 1  2 | outer.class  inner$outer.class |  **Note: You can’t directly execute the inner class’s .class file with java command.** As it is not static inner class so we can’t use static keyword with it.   How to access Inner Class Inner class can be accessed only through live instance of outer class. Within Outer Class Outer class can create instance of the inner class in the same way as normal class member.   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class OuterClass {  private int i = 9;    // Creating instance of inner class and calling inner class function  public void createInner() {  InnerClass i1 = new InnerClass();  i1.getValue();  }    // inner class declarataion  class InnerClass {  public void getValue() {  // accessing private variable from outer class  System.out.println("value of i -" + i);  }  }  } |    From Outside Outer Class Create outer class instance and then inner class instance.   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11 | class MainClass {    public static void main(String[] args) {  // Creating outer class instance  OuterClass outerclass = new OuterClass();  // Creating inner class instance  OuterClass.InnerClass innerclass = outerclass.new InnerClass();  // Classing inner class method  innerclass.getValue();  }  } |   Above code can also be replaced with   |  |  | | --- | --- | | 1 | OuterClass.InnerClass innerClass = new OuterClass.new InnerClass(); |    this keyword There are some rules associated with this and it refer the currently executing Object. So in case of Inner class “this” keyword will refer the currently executing inner class Object. But to get this for outer class use “OuterClass.this”. Modifiers Applied Normal inner class will be treated like member of the outer class so it can have several Modifiers as opposed to Class.   * + final   + abstract   + public   + private   + protected   + strictfp     Note: Don’t get confused with the modifiers of Class and Inner Class. They are completely different. Method Local Inner Class When an inner class is defined inside the method of Outer Class it becomes Method local inner class. Syntax for Creating Method Local Inner Class  |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class OuterClass {  private int i = 9;    // Creating instance of inner class and calling inner class function  public void innerMethod() {  // inner class declarataion inside method  class InnerClass {  public void getValue() {  // accessing private variable from outer class  System.out.println("value of i -" + i);  }  }  //inner class instance creation  InnerClass i1 = new InnerClass();  i1.getValue();  }  } |     Now definition of inner class is inside a method in Outer class. Still the instance of the outer class can be created but only after definition of inner class as you can see above. **Note:**  * Method local inner class can be instantiated within the method where it is defined and no where else. * Method local inner class can not use the variable defined in method where it id defined still it can use the instance variable. * If method local variable is “Final” method local inner class can use it.(\* Now variable is Final)  Modifiers Applied to Method Local Inner Class Method local inner classes are eligible for modifiers like local variable so an method local inner class can have final or abstract. | | |  | | --- | | **0. Non-Static Member Classes**    :    [**Link**](http://www.programmerinterview.com/index.php/java-questions/inner-vs-nested-classes/) |   **What is the difference between an inner and nested class in Java? What about the difference between an inner class and a static inner class?**  This tutorial is a bit complex, but we’ll try to keep things as simple as possible. With that said, let’s start off with definition and explanation of nested classes.  **Nested classes can be either static or non-static**  Nested classes can be further classified into two different types of classes: non-static nested classes and static nested classes. Non-static nested classes are more formally known as inner classes. So, think of nested classes as a big container with 2 smaller boxes inside – 1 box is for static nested classes, and another box is for inner classes (also known as non-static nested classes).  **Example of an inner class (aka non static nested class)**  Here’s a simple example of an inner class – where InnerClass is the inner class:  class OuterClass {  /\* some code here...\*/  class InnerClass { }  /\* some code here...\*/  }  Note in the code above that InnerClass is literally declared inside the OuterClass class.  **Inner classes are subsets of nested classes**   |  | | --- | |  |   Remember that an ***inner class is a specific type of nested class*** that occurs when a nested class is non-static. And that is the “difference” between an inner class and a nested class – in other words, inner classes are subsets of nested classes. So, be careful, because the terms “inner class” and “nested class” are NOT interchangeable. You can say that an inner class is also a nested class, but you can NOT say that a nested class is also an inner class. This is because nested classes are part of the larger set that includes both inner classes and static nested classes.  **What’s so special about inner classes?**  So, what exactly is special about inner classes? Well, the main thing that you must remember about inner classes is that an instance of an inner class has access to all of the members of the outer class, even those that are marked “private”. So, when an instance of the inner class is created, there are no issues with having the inner class access the instance variables of the outer class.  **Inner class versus static inner classes**  Before we dive into the differences between inner classes and static inner classes, the most important thing you should know is that *static inner classes* is the wrong terminology – they should be called *static nested classes* instead.  **Why you must use static “nested” classes instead of static “inner” classes**  There is no such thing as a static ***inner*** class, because the term “***inner*** class” means that the inner class has access to the instance variables of the outer class.  **An inner class is part of the “inner circle” of the outer class**  Think of an inner class as being part of the “inner circle” of the outer class – an inner class instance can access all the members of the outer class, even the ones declared private. But, if an inner class were also declared to be static it would be impossible to have access to all of the members of the outer class – think about why on your own before you read our answer. You should be able to come up with an answer on your own as long as you know what static means. | | |  | | --- | | **0. Local Classes and Anonymous Inner Classes Advantages and Disadvantages of using Nested/Inner class**   :    [**Link**](http://www.javaproblemstips.com/763597/) | | |  | |  |

What are the disadvantages java inner classes, the program will lead to inefficient it becomes

Category: [Java SE](http://www.javaproblemstips.com/325/1500326/)

* Access Security

* Advantage flea and tick

* Advantage Insurance

* Anonymous Class

|  |  |
| --- | --- |
| qing0910  2013-06-28 08:16:15 | java GUI programming experiments always use inner classes, this is very convenient, Internet search a bit, mostly describes the advantages of inner classes, inner classes I want to know what the shortcomings of using |
| fuhaoran520  2013-06-28 08:22:08 | There reuse reducing |

|  |  |
| --- | --- |
| sos\_hawks  2013-06-28 08:29:09 | drawback is that the class was compiled file name is difficult to see |
| skyking\_sl  2013-06-28 08:32:09 | Long-term use of inner classes drifting |

|  |  |
| --- | --- |
| w92690  2013-06-28 08:37:32 | should probably say those shortcomings is on the first floor after all, it advantage more than you |
| s6djq  2013-06-28 08:44:04 | biggest little inner classes should be able to easily access the private members of the outer class, the disadvantage is that the program structure confusing. |

|  |  |
| --- | --- |
| join330  2013-06-28 08:52:46 | inner classes can also be defined as private access security bar, valid only for this internal class |
| etc431  2013-06-28 09:00:49 | drawback I really do not know .. |

|  |  |
| --- | --- |
| clqinyi  2013-06-28 09:30:55 | decompilation of the code seems basically can not see the |
| a23800388  2013-06-28 10:03:23 | advantage that it is too ......  looks like I decompile the time, internal classes are seen clearly Well |

|  |  |
| --- | --- |
| limit13  2013-06-28 10:38:57 | similar function, if you can get through a few parameters or switches, then it is best not to use inner classes, at least not with the anonymous class |
| cs\_laker  2013-06-28 11:16:09 | you greatly. . . Thank you. . . I was java inner classes used in the trial, a little worried, afraid of causing the program efficiency, since the efficiency of the problem does not exist, then rest assured that Kazakhstan |

Domain and server ip had changed since 8/23/2013. Suspend the user registration and posts for program maintenance.

Related

* [What reasons can lead HttpUrlConnection connection time becomes longer](http://www.javaproblemstips.com/673225/)
* [Gone are slightly ~ ~ Nirvana inner class is a loud noise meter going on, what are the advantages and disadvantages shrimp](http://www.javaproblemstips.com/329855/)
* [EntityBean inefficient do?](http://www.javaproblemstips.com/360439/)
* [NIO in Selector.select () is inefficient?](http://www.javaproblemstips.com/595555/)
* [Disadvantages OSCache](http://www.javaproblemstips.com/7377/)
* [Is it only read repeatedly inefficient way to do this?](http://www.javaproblemstips.com/53869/)
* [About dblink oracle problems associated with inefficient queries](http://www.javaproblemstips.com/255700/)
* [Will: java.sql.RetsultSet in. Last () What is the mechanism of operation? This operation is inefficient.](http://www.javaproblemstips.com/399289/)
* [It becomes difficult or](http://www.javaproblemstips.com/587640/)
* [struts lead pack problem](http://www.javaproblemstips.com/8611/)

© Copyright [javaproblemstips.com](http://www.javaproblemstips.com/).[Contact Us](mailto:jamesjenny@live.com) | [Privacy Policy](http://www.javaproblemstips.com/Privacy%20Policy.html) | [Disclaimer](http://www.javaproblemstips.com/disclaimer.html) | [Disclosure](http://www.javaproblemstips.com/disclosure.html)

**Chapter 20:JDBC**

In this tutorial you will learn how to create simple JDBC application,it will show you how to open database connection, execute a SQL query and display the results. You will learn what are the different types of JDBC drivers, how to fire db queries using Prepared Statement and Callable Statement, the different types of ResultSets,using batch updates and handling transactions

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  | | --- | --- | | |  | | --- | | **0. JDBC Tutorial : Database JDBC Driver Types JDBC Architecture JDBC Types 7 Basic steps in using JDBC Three Execute methods PreparedStatment (Pre-compiled Queries) CallableStatment Metadata from DB Batch Updates ResultSet Types Transaction**   :    [**Link**](http://docs.oracle.com/javase/tutorial/jdbc/overview/)  [**Link**](http://www.tutorialspoint.com/jdbc/jdbc-statements.htm)  [**Link**](http://www.javatpoint.com/java-jdbc)  [**Link**](http://docs.oracle.com/javase/tutorial/jdbc/basics/retrieving.html) [**Link**](http://www.tutorialspoint.com/jdbc/jdbc-batch-processing.htm)  [**Link**](http://www.java2novice.com/jdbc/database-metadata/) | | |  |

# Lesson: JDBC Introduction

The JDBC API is a Java API that can access any kind of tabular data, especially data stored in a [Relational Database.](http://docs.oracle.com/javase/tutorial/jdbc/overview/#relational)

JDBC helps you to write Java applications that manage these three programming activities:

1. Connect to a data source, like a database
2. Send queries and update statements to the database
3. Retrieve and process the results received from the database in answer to your query

The following simple code fragment gives a simple example of these three steps:

public void connectToAndQueryDatabase(String username, String password) {

Connection con = DriverManager.getConnection(

"jdbc:myDriver:myDatabase",

username,

password);

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");

while (rs.next()) {

int x = rs.getInt("a");

String s = rs.getString("b");

float f = rs.getFloat("c");

}

}

This short code fragment instantiates a DriverManager object to connect to a database driver and log into the database, instantiates a Statement object that carries your SQL language query to the database; instantiates a ResultSet object that retrieves the results of your query, and executes a simple while loop, which retrieves and displays those results. It's that simple.

## JDBC Product Components

JDBC includes four components:

1. **The JDBC API** —  The JDBC™ API provides programmatic access to relational data from the Java™ programming language. Using the JDBC API, applications can execute SQL statements, retrieve results, and propagate changes back to an underlying data source. The JDBC API can also interact with multiple data sources in a distributed, heterogeneous environment.

The JDBC API is part of the Java platform, which includes the *Java™ Standard Edition* (Java™ SE ) and the *Java™ Enterprise Edition* (Java™ EE). The JDBC 4.0 API is divided into two packages: java.sql and javax.sql. Both packages are included in the Java SE and Java EE platforms.

1. **JDBC Driver Manager** —  The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager has traditionally been the backbone of the JDBC architecture. It is quite small and simple.

The Standard Extension packages javax.naming and javax.sql let you use a DataSource object registered with a *Java Naming and Directory Interface*™ (JNDI) naming service to establish a connection with a data source. You can use either connecting mechanism, but using a DataSource object is recommended whenever possible.

1. **JDBC Test Suite** —  The JDBC driver test suite helps you to determine that JDBC drivers will run your program. These tests are not comprehensive or exhaustive, but they do exercise many of the important features in the JDBC API.
2. **JDBC-ODBC Bridge** —  The Java Software bridge provides JDBC access via ODBC drivers. Note that you need to load ODBC binary code onto each client machine that uses this driver. As a result, the ODBC driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in a three-tier architecture.

This Trail uses the first two of these these four JDBC components to connect to a database and then build a java program that uses SQL commands to communicate with a test Relational Database. The last two components are used in specialized environments to test web applications, or to communicate with ODBC-aware DBMSs.

## JDBC Architecture

### Two-tier and Three-tier Processing Models

The JDBC API supports both two-tier and three-tier processing models for database access.

***Figure 1: Two-tier Architecture for Data Access.***



In the two-tier model, a Java applet or application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

***Figure 2: Three-tier Architecture for Data Access.***



Until recently, the middle tier has often been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java bytecode into efficient machine-specific code and technologies such as Enterprise JavaBeans™, the Java platform is fast becoming the standard platform for middle-tier development. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features.

With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture. Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected rowsets. The JDBC API is also what allows access to a data source from a Java middle tier.

## A Relational Database Overview

A database is a means of storing information in such a way that information can be retrieved from it. In simplest terms, a relational database is one that presents information in tables with rows and columns. A table is referred to as a relation in the sense that it is a collection of objects of the same type (rows). Data in a table can be related according to common keys or concepts, and the ability to retrieve related data from a table is the basis for the term relational database. A Database Management System (DBMS) handles the way data is stored, maintained, and retrieved. In the case of a relational database, a Relational Database Management System (RDBMS) performs these tasks. DBMS as used in this book is a general term that includes RDBMS.

### Integrity Rules

Relational tables follow certain integrity rules to ensure that the data they contain stay accurate and are always accessible. First, the rows in a relational table should all be distinct. If there are duplicate rows, there can be problems resolving which of two possible selections is the correct one. For most DBMSs, the user can specify that duplicate rows are not allowed, and if that is done, the DBMS will prevent the addition of any rows that duplicate an existing row.

A second integrity rule of the traditional relational model is that column values must not be repeating groups or arrays. A third aspect of data integrity involves the concept of a null value. A database takes care of situations where data may not be available by using a null value to indicate that a value is missing. It does not equate to a blank or zero. A blank is considered equal to another blank, a zero is equal to another zero, but two null values are not considered equal.

When each row in a table is different, it is possible to use one or more columns to identify a particular row. This unique column or group of columns is called a primary key. Any column that is part of a primary key cannot be null; if it were, the primary key containing it would no longer be a complete identifier. This rule is referred to as entity integrity.

The Employees table illustrates some of these relational database concepts. It has five columns and six rows, with each row representing a different employee.

Employees Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Employee\_Number** | **First\_name** | **Last\_Name** | **Date\_of\_Birth** | **Car\_Number** |
| 10001 | Axel | Washington | 28-Aug-43 | 5 |
| 10083 | Arvid | Sharma | 24-Nov-54 | null |
| 10120 | Jonas | Ginsberg | 01-Jan-69 | null |
| 10005 | Florence | Wojokowski | 04-Jul-71 | 12 |
| 10099 | Sean | Washington | 21-Sep-66 | null |
| 10035 | Elizabeth | Yamaguchi | 24-Dec-59 | null |

The primary key for this table would generally be the employee number because each one is guaranteed to be different. (A number is also more efficient than a string for making comparisons.) It would also be possible to use First\_Name and Last\_Name because the combination of the two also identifies just one row in our sample database. Using the last name alone would not work because there are two employees with the last name of "Washington." In this particular case the first names are all different, so one could conceivably use that column as a primary key, but it is best to avoid using a column where duplicates could occur. If Elizabeth Yamaguchi gets a job at this company and the primary key is First\_Name, the RDBMS will not allow her name to be added (if it has been specified that no duplicates are permitted). Because there is already an Elizabeth in the table, adding a second one would make the primary key useless as a way of identifying just one row. Note that although using First\_Name and Last\_Name is a unique composite key for this example, it might not be unique in a larger database. Note also that the Employee table assumes that there can be only one car per employee.

### SELECT Statements

SQL is a language designed to be used with relational databases. There is a set of basic SQL commands that is considered standard and is used by all RDBMSs. For example, all RDBMSs use the SELECT statement.

A SELECT statement, also called a query, is used to get information from a table. It specifies one or more column headings, one or more tables from which to select, and some criteria for selection. The RDBMS returns rows of the column entries that satisfy the stated requirements. A SELECT statement such as the following will fetch the first and last names of employees who have company cars:

SELECT First\_Name, Last\_Name

FROM Employees

WHERE Car\_Number IS NOT NULL

The result set (the set of rows that satisfy the requirement of not having null in the Car\_Number column) follows. The first name and last name are printed for each row that satisfies the requirement because the SELECT statement (the first line) specifies the columns First\_Name and Last\_Name. The FROM clause (the second line) gives the table from which the columns will be selected.

|  |  |
| --- | --- |
| **FIRST\_NAME** | **LAST\_NAME** |
| Axel | Washington |
| Florence | Wojokowski |

The following code produces a result set that includes the whole table because it asks for all of the columns in the table Employees with no restrictions (no WHERE clause). Note that SELECT \* means "SELECT all columns."

SELECT \*

FROM Employees

### WHERE Clauses

The WHERE clause in a SELECT statement provides the criteria for selecting values. For example, in the following code fragment, values will be selected only if they occur in a row in which the column Last\_Name begins with the string 'Washington'.

SELECT First\_Name, Last\_Name

FROM Employees

WHERE Last\_Name LIKE 'Washington%'

The keyword LIKE is used to compare strings, and it offers the feature that patterns containing wildcards can be used. For example, in the code fragment above, there is a percent sign (%) at the end of 'Washington', which signifies that any value containing the string 'Washington' plus zero or more additional characters will satisfy this selection criterion. So 'Washington' or 'Washingtonian' would be matches, but 'Washing' would not be. The other wildcard used in LIKE clauses is an underbar (\_), which stands for any one character. For example,

WHERE Last\_Name LIKE 'Ba\_man'

would match 'Batman', 'Barman', 'Badman', 'Balman', 'Bagman', 'Bamman', and so on.

The code fragment below has a WHERE clause that uses the equal sign (=) to compare numbers. It selects the first and last name of the employee who is assigned car 12.

SELECT First\_Name, Last\_Name

FROM Employees

WHERE Car\_Number = 12

The next code fragment selects the first and last names of employees whose employee number is greater than 10005:

SELECT First\_Name, Last\_Name

FROM Employees

WHERE Employee\_Number > 10005

WHERE clauses can get rather elaborate, with multiple conditions and, in some DBMSs, nested conditions. This overview will not cover complicated WHERE clauses, but the following code fragment has a WHERE clause with two conditions; this query selects the first and last names of employees whose employee number is less than 10100 and who do not have a company car.

SELECT First\_Name, Last\_Name

FROM Employees

WHERE Employee\_Number < 10100 and Car\_Number IS NULL

A special type of WHERE clause involves a join, which is explained in the next section.

### Joins

A distinguishing feature of relational databases is that it is possible to get data from more than one table in what is called a join. Suppose that after retrieving the names of employees who have company cars, one wanted to find out who has which car, including the make, model, and year of car. This information is stored in another table, Cars:

Cars Table

|  |  |  |  |
| --- | --- | --- | --- |
| **Car\_Number** | **Make** | **Model** | **Year** |
| 5 | Honda | Civic DX | 1996 |
| 12 | Toyota | Corolla | 1999 |

There must be one column that appears in both tables in order to relate them to each other. This column, which must be the primary key in one table, is called the foreign key in the other table. In this case, the column that appears in two tables is Car\_Number, which is the primary key for the table Cars and the foreign key in the table Employees. If the 1996 Honda Civic were wrecked and deleted from the Cars table, then Car\_Number 5 would also have to be removed from the Employees table in order to maintain what is called referential integrity. Otherwise, the foreign key column (Car\_Number) in the Employees table would contain an entry that did not refer to anything in Cars. A foreign key must either be null or equal to an existing primary key value of the table to which it refers. This is different from a primary key, which may not be null. There are several null values in the Car\_Number column in the table Employees because it is possible for an employee not to have a company car.

The following code asks for the first and last names of employees who have company cars and for the make, model, and year of those cars. Note that the FROM clause lists both Employees and Cars because the requested data is contained in both tables. Using the table name and a dot (.) before the column name indicates which table contains the column.

SELECT Employees.First\_Name, Employees.Last\_Name,

Cars.Make, Cars.Model, Cars.Year

FROM Employees, Cars

WHERE Employees.Car\_Number = Cars.Car\_Number

This returns a result set that will look similar to the following:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **FIRST\_NAME** | **LAST\_NAME** | **MAKE** | **MODEL** | **YEAR** |
| Axel | Washington | Honda | Civic DX | 1996 |
| Florence | Wojokowski | Toyota | Corolla | 1999 |

### Common SQL Commands

SQL commands are divided into categories, the two main ones being Data Manipulation Language (DML) commands and Data Definition Language (DDL) commands. DML commands deal with data, either retrieving it or modifying it to keep it up-to-date. DDL commands create or change tables and other database objects such as views and indexes.

A list of the more common DML commands follows:

* SELECT —  used to query and display data from a database. The SELECT statement specifies which columns to include in the result set. The vast majority of the SQL commands used in applications are SELECT statements.
* INSERT —  adds new rows to a table. INSERT is used to populate a newly created table or to add a new row (or rows) to an already-existing table.
* DELETE —  removes a specified row or set of rows from a table
* UPDATE —  changes an existing value in a column or group of columns in a table

The more common DDL commands follow:

* CREATE TABLE —  creates a table with the column names the user provides. The user also needs to specify a type for the data in each column. Data types vary from one RDBMS to another, so a user might need to use metadata to establish the data types used by a particular database. CREATE TABLE is normally used less often than the data manipulation commands because a table is created only once, whereas adding or deleting rows or changing individual values generally occurs more frequently.
* DROP TABLE —  deletes all rows and removes the table definition from the database. A JDBC API implementation is required to support the DROP TABLE command as specified by SQL92, Transitional Level. However, support for the CASCADE and RESTRICT options of DROP TABLE is optional. In addition, the behavior of DROP TABLEis implementation-defined when there are views or integrity constraints defined that reference the table being dropped.
* ALTER TABLE —  adds or removes a column from a table. It also adds or drops table constraints and alters column attributes

### Result Sets and Cursors

The rows that satisfy the conditions of a query are called the result set. The number of rows returned in a result set can be zero, one, or many. A user can access the data in a result set one row at a time, and a cursor provides the means to do that. A cursor can be thought of as a pointer into a file that contains the rows of the result set, and that pointer has the ability to keep track of which row is currently being accessed. A cursor allows a user to process each row of a result set from top to bottom and consequently may be used for iterative processing. Most DBMSs create a cursor automatically when a result set is generated.

Earlier JDBC API versions added new capabilities for a result set's cursor, allowing it to move both forward and backward and also allowing it to move to a specified row or to a row whose position is relative to another row.

### Transactions

When one user is accessing data in a database, another user may be accessing the same data at the same time. If, for instance, the first user is updating some columns in a table at the same time the second user is selecting columns from that same table, it is possible for the second user to get partly old data and partly updated data. For this reason, DBMSs use transactions to maintain data in a consistent state (data consistency) while allowing more than one user to access a database at the same time (data concurrency).

A transaction is a set of one or more SQL statements that make up a logical unit of work. A transaction ends with either a commit or a rollback, depending on whether there are any problems with data consistency or data concurrency. The commit statement makes permanent the changes resulting from the SQL statements in the transaction, and the rollback statement undoes all changes resulting from the SQL statements in the transaction.

A lock is a mechanism that prohibits two transactions from manipulating the same data at the same time. For example, a table lock prevents a table from being dropped if there is an uncommitted transaction on that table. In some DBMSs, a table lock also locks all of the rows in a table. A row lock prevents two transactions from modifying the same row, or it prevents one transaction from selecting a row while another transaction is still modifying it.

### Stored Procedures

A stored procedure is a group of SQL statements that can be called by name. In other words, it is executable code, a mini-program, that performs a particular task that can be invoked the same way one can call a function or method. Traditionally, stored procedures have been written in a DBMS-specific programming language. The latest generation of database products allows stored procedures to be written using the Java programming language and the JDBC API. Stored procedures written in the Java programming language are bytecode portable between DBMSs. Once a stored procedure is written, it can be used and reused because a DBMS that supports stored procedures will, as its name implies, store it in the database.

The following code is an example of how to create a very simple stored procedure using the Java programming language. Note that the stored procedure is just a static Java method that contains normal JDBC code. It accepts two input parameters and uses them to change an employee's car number.

Do not worry if you do not understand the example at this point. The code example below is presented only to illustrate what a stored procedure looks like. You will learn how to write the code in this example in the tutorials that follow.

import java.sql.\*;

public class UpdateCar {

public static void UpdateCarNum(int carNo, int empNo)

throws SQLException {

Connection con = null;

PreparedStatement pstmt = null;

try {

con = DriverManager.getConnection(

"jdbc:default:connection");

pstmt = con.prepareStatement(

"UPDATE EMPLOYEES " +

"SET CAR\_NUMBER = ? " +

"WHERE EMPLOYEE\_NUMBER = ?");

pstmt.setInt(1, carNo);

pstmt.setInt(2, empNo);

pstmt.executeUpdate();

}

finally {

if (pstmt != null) pstmt.close();

}

}

}

### Metadata

Databases store user data, and they also store information about the database itself. Most DBMSs have a set of system tables, which list tables in the database, column names in each table, primary keys, foreign keys, stored procedures, and so forth. Each DBMS has its own functions for getting information about table layouts and database features. JDBC provides the interface DatabaseMetaData, which a driver writer must implement so that its methods return information about the driver and/or DBMS for which the driver is written. For example, a large number of methods return whether or not the driver supports a particular functionality. This interface gives users and tools a standardized way to get metadata.

In general, developers writing tools and drivers are the ones most likely to be concerned with metadata.

Once a connection is obtained we can interact with the database. The JDBC *Statement, CallableStatement,* and *PreparedStatement* interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

They also define methods that help bridge data type differences between Java and SQL data types used in a database.

The following table provides a summary of each interface's purpose to decide on the interface to use.

|  |  |
| --- | --- |
| **Interfaces** | **Recommended Use** |
| Statement | Use the for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters. |
| PreparedStatement | Use the when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime. |
| CallableStatement | Use the when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters. |

## The Statement Objects

### Creating Statement Object

Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's createStatement( ) method, as in the following example −

Statement stmt = null;

try {

stmt = conn.createStatement( );

. . .

}

catch (SQLException e) {

. . .

}

finally {

. . .

}

Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods.

* **boolean execute (String SQL)**: Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
* **int executeUpdate (String SQL)**: Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
* **ResultSet executeQuery (String SQL)**: Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

## Closing Statement Object

Just as you close a Connection object to save database resources, for the same reason you should also close the Statement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the Statement object as well. However, you should always explicitly close the Statement object to ensure proper cleanup.

Statement stmt = null;

try {

stmt = conn.createStatement( );

. . .

}

catch (SQLException e) {

. . .

}

finally {

stmt.close();

}

For a better understanding, we suggest you to study the [**Statement - Example tutorial**](http://www.tutorialspoint.com/jdbc/statement-object-example.htm).

## The PreparedStatement Objects

The *PreparedStatement* interface extends the Statement interface, which gives you added functionality with a couple of advantages over a generic Statement object.

This statement gives you the flexibility of supplying arguments dynamically.

## Creating PreparedStatement Object

PreparedStatement pstmt = null;

try {

String SQL = "Update Employees SET age = ? WHERE id = ?";

pstmt = conn.prepareStatement(SQL);

. . .

}

catch (SQLException e) {

. . .

}

finally {

. . .

}

All parameters in JDBC are represented by the **?** symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException.

Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0.

All of the **Statement object's** methods for interacting with the database (a) execute(), (b) executeQuery(), and (c) executeUpdate() also work with the PreparedStatement object. However, the methods are modified to use SQL statements that can input the parameters.

## Closing PreparedStatement Object

Just as you close a Statement object, for the same reason you should also close the PreparedStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the PreparedStatement object as well. However, you should always explicitly close the PreparedStatement object to ensure proper cleanup.

PreparedStatement pstmt = null;

try {

String SQL = "Update Employees SET age = ? WHERE id = ?";

pstmt = conn.prepareStatement(SQL);

. . .

}

catch (SQLException e) {

. . .

}

finally {

pstmt.close();

}

For a better understanding, let us study [**Prepare - Example Code**](http://www.tutorialspoint.com/jdbc/preparestatement-object-example.htm).

## The CallableStatement Objects

Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object, which would be used to execute a call to a database stored procedure.

## Creating CallableStatement Object

Suppose, you need to execute the following Oracle stored procedure −

CREATE OR REPLACE PROCEDURE getEmpName

(EMP\_ID IN NUMBER, EMP\_FIRST OUT VARCHAR) AS

BEGIN

SELECT first INTO EMP\_FIRST

FROM Employees

WHERE ID = EMP\_ID;

END;

**NOTE:** Above stored procedure has been written for Oracle, but we are working with MySQL database so, let us write same stored procedure for MySQL as follows to create it in EMP database −

DELIMITER $$

DROP PROCEDURE IF EXISTS `EMP`.`getEmpName` $$

CREATE PROCEDURE `EMP`.`getEmpName`

(IN EMP\_ID INT, OUT EMP\_FIRST VARCHAR(255))

BEGIN

SELECT first INTO EMP\_FIRST

FROM Employees

WHERE ID = EMP\_ID;

END $$

DELIMITER ;

Three types of parameters exist: IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all the three.

Here are the definitions of each −

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| IN | A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods. |
| OUT | A parameter whose value is supplied by the SQL statement it returns. You retrieve values from theOUT parameters with the getXXX() methods. |
| INOUT | A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods. |

The following code snippet shows how to employ the **Connection.prepareCall()** method to instantiate a **CallableStatement** object based on the preceding stored procedure −

CallableStatement cstmt = null;

try {

String SQL = "{call getEmpName (?, ?)}";

cstmt = conn.prepareCall (SQL);

. . .

}

catch (SQLException e) {

. . .

}

finally {

. . .

}

The String variable SQL, represents the stored procedure, with parameter placeholders.

Using the CallableStatement objects is much like using the PreparedStatement objects. You must bind values to all the parameters before executing the statement, or you will receive an SQLException.

If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the setXXX() method that corresponds to the Java data type you are binding.

When you use OUT and INOUT parameters you must employ an additional CallableStatement method, registerOutParameter(). The registerOutParameter() method binds the JDBC data type, to the data type that the stored procedure is expected to return.

Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate getXXX() method. This method casts the retrieved value of SQL type to a Java data type.

## Closing CallableStatement Object

Just as you close other Statement object, for the same reason you should also close the CallableStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the CallableStatement object as well. However, you should always explicitly close the CallableStatement object to ensure proper cleanup.

CallableStatement cstmt = null;

try {

String SQL = "{call getEmpName (?, ?)}";

cstmt = conn.prepareCall (SQL);

. . .

}

catch (SQLException e) {

. . .

}

finally {

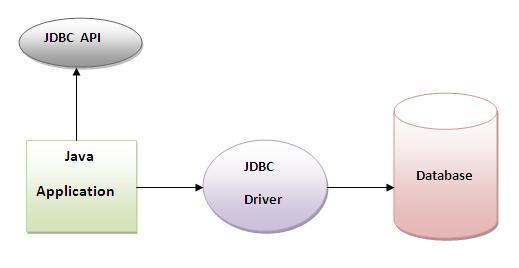
cstmt.close();

}

For a better understanding, I would suggest to study [**Callable - Example Code**](http://www.tutorialspoint.com/jdbc/callablestatement-object-example.htm).

# Java JDBC Tutorial

Java JDBC is a java API to connect and execute query with the database. JDBC API uses jdbc drivers to connect with the database.



### Why use JDBC

Before JDBC, ODBC API was the database API to connect and execute query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

Do You Know

* How to connect Java application with Oracle and Mysql database using JDBC?
* What is the difference between Statement and PreparedStatement interface?
* How to print total numbers of tables and views of a database using JDBC ?
* How to store and retrieve images from Oracle database using JDBC?
* How to store and retrieve files from Oracle database using JDBC?

## What is API

API (Application programming interface) is a document that contains description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc

## Topics in Java JDBC Tutorial

[2) JDBC Drivers](http://www.javatpoint.com/jdbc-driver)

In this JDBC tutorial, we will learn 4 types of JDBC drivers, their advantages and disadvantages.

[3) 5 Steps to connect to the database](http://www.javatpoint.com/steps-to-connect-to-the-database-in-java)

In this JDBC tutorial, we will see the 5 steps to connect to the database in java using JDBC.

[4) Connectivity with Oracle using JDBC](http://www.javatpoint.com/example-to-connect-to-the-oracle-database)

In this JDBC tutorial, we will connect a simple java program with the oracle database.

[5) Connectivity with MySQL using JDBC](http://www.javatpoint.com/example-to-connect-to-the-mysql-database)

In this JDBC tutorial, we will connect a simple java program with the mysql database.

[6) Connectivity with Access without DSN](http://www.javatpoint.com/connectivity-with-access-without-dsn)

Let's connect java application with access database with and without DSN.

[7) DriverManager class](http://www.javatpoint.com/DriverManager-class)

In this JDBC tutorial, we will learn what does the DriverManager class and what are its methods.

[8) Connection interface](http://www.javatpoint.com/Connection-interface)

In this JDBC tutorial, we will learn what is Connection interface and what are its methods.

[9) Statement interface](http://www.javatpoint.com/Statement-interface)

In this JDBC tutorial, we will learn what is Statement interface and what are its methods.

[10) ResultSet interface](http://www.javatpoint.com/ResultSet-interface)

In this JDBC tutorial, we will learn what is ResultSet interface and what are its methods. Moreover, we will learn how we can make the ResultSet scrollable.

[11) PreparedStatement Interface](http://www.javatpoint.com/PreparedStatement-interface)

In this JDBC tutorial, we will learn what is benefit of PreparedStatement over Statement interface. We will see examples to insert, update or delete records using the PreparedStatement interface.

[12) ResultSetMetaData interface](http://www.javatpoint.com/ResultSetMetaData-interface)

In this JDBC tutorial, we will learn how we can get the metadata of a table.

[13) DatabaseMetaData interface](http://www.javatpoint.com/DatabaseMetaData-interface)

In this JDBC tutorial, we will learn how we can get the metadata of a database.

[14) Storing image in Oracle](http://www.javatpoint.com/storing-image-in-oracle-database)

Let's learn how to store image in the oracle database using JDBC.

[15) Retrieving image from Oracle](http://www.javatpoint.com/retrieving-image-from-oracle-database)

Let's see the simple example to retrieve image from the oracle database using JDBC.

[16) Storing file in Oracle](http://www.javatpoint.com/storing-file-in-oracle-database)

Let's see the simple example to store file in the oracle database using JDBC.

[17) Retrieving file from Oracle](http://www.javatpoint.com/retrieving-file-from-oracle-database)

Let's see the simple example to retrieve file from the oracle database using JDBC.

[18) CallableStatement](http://www.javatpoint.com/CallableStatement-interface)

Let's see the code to call stored procedures and functions using CallableStatement.

[19) Transaction Management using JDBC](http://www.javatpoint.com/transaction-management-in-jdbc)

Let's see the simple example to use transaction management using JDBC.

[20) Batch Statement using JDBC](http://www.javatpoint.com/batch-processing-in-jdbc)

Let's see the code to execute batch of queries.

[21) JDBC RowSet](http://www.javatpoint.com/jdbc-rowset)

Let's see the working of new JDBC RowSet interface.

# Retrieving and Modifying Values from Result Sets

The following method, [CoffeesTable.viewTable](http://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html) outputs the contents of the COFFEES tables, and demonstrates the use of ResultSet objects and cursors:

public static void viewTable(Connection con, String dbName)

throws SQLException {

Statement stmt = null;

String query =

"select COF\_NAME, SUP\_ID, PRICE, " +

"SALES, TOTAL " +

"from " + dbName + ".COFFEES";

try {

stmt = con.createStatement();

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) {

String coffeeName = rs.getString("COF\_NAME");

int supplierID = rs.getInt("SUP\_ID");

float price = rs.getFloat("PRICE");

int sales = rs.getInt("SALES");

int total = rs.getInt("TOTAL");

System.out.println(coffeeName + "\t" + supplierID +

"\t" + price + "\t" + sales +

"\t" + total);

}

} catch (SQLException e ) {

JDBCTutorialUtilities.printSQLException(e);

} finally {

if (stmt != null) { stmt.close(); }

}

}

A ResultSet object is a table of data representing a database result set, which is usually generated by executing a statement that queries the database. For example, the[CoffeeTables.viewTable](http://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html) method creates a ResultSet, rs, when it executes the query through the Statement object, stmt. Note that a ResultSet object can be created through any object that implements the Statement interface, including PreparedStatement, CallableStatement, and RowSet.

You access the data in a ResultSet object through a cursor. Note that this cursor is not a database cursor. This cursor is a pointer that points to one row of data in theResultSet. Initially, the cursor is positioned before the first row. The method ResultSet.next moves the cursor to the next row. This method returns false if the cursor is positioned after the last row. This method repeatedly calls the ResultSet.next method with a while loop to iterate through all the data in the ResultSet.

This page covers the following topics:

* [ResultSet Interface](http://docs.oracle.com/javase/tutorial/jdbc/basics/retrieving.html#rs_interface)
* [Retrieving Column Values from Rows](http://docs.oracle.com/javase/tutorial/jdbc/basics/retrieving.html#retrieve_rs)
* [Cursors](http://docs.oracle.com/javase/tutorial/jdbc/basics/retrieving.html#cursors)
* [Updating Rows in ResultSet Objects](http://docs.oracle.com/javase/tutorial/jdbc/basics/retrieving.html#rs_update)
* [Using Statement Objects for Batch Updates](http://docs.oracle.com/javase/tutorial/jdbc/basics/retrieving.html#batch_updates)
* [Inserting Rows in ResultSet Objects](http://docs.oracle.com/javase/tutorial/jdbc/basics/retrieving.html#rs_insert)

## ResultSet Interface

The ResultSet interface provides methods for retrieving and manipulating the results of executed queries, and ResultSet objects can have different functionality and characteristics. These characteristics are type, concurrency, and cursor holdability.

### ResultSet Types

The type of a ResultSet object determines the level of its functionality in two areas: the ways in which the cursor can be manipulated, and how concurrent changes made to the underlying data source are reflected by the ResultSet object.

The sensitivity of a ResultSet object is determined by one of three different ResultSet types:

* TYPE\_FORWARD\_ONLY: The result set cannot be scrolled; its cursor moves forward only, from before the first row to after the last row. The rows contained in the result set depend on how the underlying database generates the results. That is, it contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.
* TYPE\_SCROLL\_INSENSITIVE: The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set is insensitive to changes made to the underlying data source while it is open. It contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.
* TYPE\_SCROLL\_SENSITIVE: The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set reflects changes made to the underlying data source while the result set remains open.

The default ResultSet type is TYPE\_FORWARD\_ONLY.

**Note**: Not all databases and JDBC drivers support all ResultSet types. The method DatabaseMetaData.supportsResultSetType returns true if the specifiedResultSet type is supported and false otherwise.

### ResultSet Concurrency

The concurrency of a ResultSet object determines what level of update functionality is supported.

There are two concurrency levels:

* CONCUR\_READ\_ONLY: The ResultSet object cannot be updated using the ResultSet interface.
* CONCUR\_UPDATABLE: The ResultSet object can be updated using the ResultSet interface.

The default ResultSet concurrency is CONCUR\_READ\_ONLY.

**Note**: Not all JDBC drivers and databases support concurrency. The method DatabaseMetaData.supportsResultSetConcurrency returns true if the specified concurrency level is supported by the driver and false otherwise.

The method [CoffeesTable.modifyPrices](http://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html) demonstrates how to use a ResultSet object whose concurrency level is CONCUR\_UPDATABLE.

### Cursor Holdability

Calling the method Connection.commit can close the ResultSet objects that have been created during the current transaction. In some cases, however, this may not be the desired behavior. The ResultSet property holdability gives the application control over whether ResultSet objects (cursors) are closed when commit is called.

The following ResultSet constants may be supplied to the Connection methods createStatement, prepareStatement, and prepareCall:

* HOLD\_CURSORS\_OVER\_COMMIT: ResultSet cursors are not closed; they are holdable: they are held open when the method commit is called. Holdable cursors might be ideal if your application uses mostly read-only ResultSet objects.
* CLOSE\_CURSORS\_AT\_COMMIT: ResultSet objects (cursors) are closed when the commit method is called. Closing cursors when this method is called can result in better performance for some applications.

The default cursor holdability varies depending on your DBMS.

**Note**: Not all JDBC drivers and databases support holdable and non-holdable cursors. The following method, JDBCTutorialUtilities.cursorHoldabilitySupport, outputs the default cursor holdability of ResultSet objects and whether HOLD\_CURSORS\_OVER\_COMMIT and CLOSE\_CURSORS\_AT\_COMMIT are supported:

public static void cursorHoldabilitySupport(Connection conn)

throws SQLException {

DatabaseMetaData dbMetaData = conn.getMetaData();

System.out.println("ResultSet.HOLD\_CURSORS\_OVER\_COMMIT = " +

ResultSet.HOLD\_CURSORS\_OVER\_COMMIT);

System.out.println("ResultSet.CLOSE\_CURSORS\_AT\_COMMIT = " +

ResultSet.CLOSE\_CURSORS\_AT\_COMMIT);

System.out.println("Default cursor holdability: " +

**dbMetaData.getResultSetHoldability()**);

System.out.println("Supports HOLD\_CURSORS\_OVER\_COMMIT? " +

**dbMetaData.supportsResultSetHoldability(**

**ResultSet.HOLD\_CURSORS\_OVER\_COMMIT)**);

System.out.println("Supports CLOSE\_CURSORS\_AT\_COMMIT? " +

**dbMetaData.supportsResultSetHoldability(**

**ResultSet.CLOSE\_CURSORS\_AT\_COMMIT)**);

}

## Retrieving Column Values from Rows

The ResultSet interface declares getter methods (for example, getBoolean and getLong) for retrieving column values from the current row. You can retrieve values using either the index number of the column or the alias or name of the column. The column index is usually more efficient. Columns are numbered from 1. For maximum portability, result set columns within each row should be read in left-to-right order, and each column should be read only once.

For example, the following method, [CoffeesTable.alternateViewTable](http://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html), retrieves column values by number:

public static void alternateViewTable(Connection con)

throws SQLException {

Statement stmt = null;

String query =

"select COF\_NAME, SUP\_ID, PRICE, " +

"SALES, TOTAL from COFFEES";

try {

stmt = con.createStatement();

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) {

String coffeeName = rs.getString(1);

int supplierID = rs.getInt(2);

float price = rs.getFloat(3);

int sales = rs.getInt(4);

int total = rs.getInt(5);

System.out.println(coffeeName + "\t" + supplierID +

"\t" + price + "\t" + sales +

"\t" + total);

}

} catch (SQLException e ) {

JDBCTutorialUtilities.printSQLException(e);

} finally {

if (stmt != null) { stmt.close(); }

}

}

Strings used as input to getter methods are case-insensitive. When a getter method is called with a string and more than one column has the same alias or name as the string, the value of the first matching column is returned. The option to use a string as opposed to an integer is designed to be used when column aliases and names are used in the SQL query that generated the result set. For columns that are not explicitly named in the query (for example, select \* from COFFEES) it is best to use column numbers. If column names are used, the developer should guarantee that they uniquely refer to the intended columns by using column aliases. A column alias effectively renames the column of a result set. To specify a column alias, use the SQL AS clause in the SELECT statement.

The getter method of the appropriate type retrieves the value in each column. For example, in the method [CoffeeTables.viewTable](http://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html), the first column in each row of theResultSet rs is COF\_NAME, which stores a value of SQL type VARCHAR. The method for retrieving a value of SQL type VARCHAR is getString. The second column in each row stores a value of SQL type INTEGER, and the method for retrieving values of that type is getInt.

Note that although the method getString is recommended for retrieving the SQL types CHAR and VARCHAR, it is possible to retrieve any of the basic SQL types with it. Getting all values with getString can be very useful, but it also has its limitations. For instance, if it is used to retrieve a numeric type, getString converts the numeric value to a Java String object, and the value has to be converted back to a numeric type before it can be operated on as a number. In cases where the value is treated as a string anyway, there is no drawback. Furthermore, if you want an application to retrieve values of any standard SQL type other than SQL3 types, use the getString method.

## Cursors

As mentioned previously, you access the data in a ResultSet object through a cursor, which points to one row in the ResultSet object. However, when a ResultSet object is first created, the cursor is positioned before the first row. The method [CoffeeTables.viewTable](http://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html) moves the cursor by calling the ResultSet.next method. There are other methods available to move the cursor:

* next: Moves the cursor forward one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned after the last row.
* previous: Moves the cursor backward one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned before the first row.
* first: Moves the cursor to the first row in the ResultSet object. Returns true if the cursor is now positioned on the first row and false if the ResultSet object does not contain any rows.
* last:: Moves the cursor to the last row in the ResultSet object. Returns true if the cursor is now positioned on the last row and false if the ResultSet object does not contain any rows.
* beforeFirst: Positions the cursor at the start of the ResultSet object, before the first row. If the ResultSet object does not contain any rows, this method has no effect.
* afterLast: Positions the cursor at the end of the ResultSet object, after the last row. If the ResultSet object does not contain any rows, this method has no effect.
* relative(int rows): Moves the cursor relative to its current position.
* absolute(int row): Positions the cursor on the row specified by the parameter row.

Note that the default sensitivity of a ResultSet is TYPE\_FORWARD\_ONLY, which means that it cannot be scrolled; you cannot call any of these methods that move the cursor, except next, if your ResultSet cannot be scrolled. The method [CoffeesTable.modifyPrices](http://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html), described in the following section, demonstrates how you can move the cursor of a ResultSet.

## Updating Rows in ResultSet Objects

You cannot update a default ResultSet object, and you can only move its cursor forward. However, you can create ResultSet objects that can be scrolled (the cursor can move backwards or move to an absolute position) and updated.

The following method, [CoffeesTable.modifyPrices](http://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html), multiplies the PRICE column of each row by the argument percentage:

public void modifyPrices(float percentage) throws SQLException {

Statement stmt = null;

try {

stmt = con.createStatement();

stmt = con.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE,

ResultSet.CONCUR\_UPDATABLE);

ResultSet uprs = stmt.executeQuery(

"SELECT \* FROM " + dbName + ".COFFEES");

while (uprs.next()) {

float f = uprs.getFloat("PRICE");

uprs.updateFloat( "PRICE", f \* percentage);

uprs.updateRow();

}

} catch (SQLException e ) {

JDBCTutorialUtilities.printSQLException(e);

} finally {

if (stmt != null) { stmt.close(); }

}

}

The field ResultSet.TYPE\_SCROLL\_SENSITIVE creates a ResultSet object whose cursor can move both forward and backward relative to the current position and to an absolute position. The field ResultSet.CONCUR\_UPDATABLE creates a ResultSet object that can be updated. See the ResultSet Javadoc for other fields you can specify to modify the behavior of ResultSet objects.

The method ResultSet.updateFloat updates the specified column (in this example, PRICE with the specified float value in the row where the cursor is positioned.ResultSet contains various updater methods that enable you to update column values of various data types. However, none of these updater methods modifies the database; you must call the method ResultSet.updateRow to update the database.

## Using Statement Objects for Batch Updates

Statement, PreparedStatement and CallableStatement objects have a list of commands that is associated with them. This list may contain statements for updating, inserting, or deleting a row; and it may also contain DDL statements such as CREATE TABLE and DROP TABLE. It cannot, however, contain a statement that would produce aResultSet object, such as a SELECT statement. In other words, the list can contain only statements that produce an update count.

The list, which is associated with a Statement object at its creation, is initially empty. You can add SQL commands to this list with the method addBatch and empty it with the method clearBatch. When you have finished adding statements to the list, call the method executeBatch to send them all to the database to be executed as a unit, or batch.

For example, the following method [CoffeesTable.batchUpdate](http://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html) adds four rows to the COFFEES table with a batch update:

public void batchUpdate() throws SQLException {

Statement stmt = null;

try {

this.con.setAutoCommit(false);

stmt = this.con.createStatement();

stmt.addBatch(

"INSERT INTO COFFEES " +

"VALUES('Amaretto', 49, 9.99, 0, 0)");

stmt.addBatch(

"INSERT INTO COFFEES " +

"VALUES('Hazelnut', 49, 9.99, 0, 0)");

stmt.addBatch(

"INSERT INTO COFFEES " +

"VALUES('Amaretto\_decaf', 49, " +

"10.99, 0, 0)");

stmt.addBatch(

"INSERT INTO COFFEES " +

"VALUES('Hazelnut\_decaf', 49, " +

"10.99, 0, 0)");

int [] updateCounts = stmt.executeBatch();

this.con.commit();

} catch(BatchUpdateException b) {

JDBCTutorialUtilities.printBatchUpdateException(b);

} catch(SQLException ex) {

JDBCTutorialUtilities.printSQLException(ex);

} finally {

if (stmt != null) { stmt.close(); }

this.con.setAutoCommit(true);

}

}

The following line disables auto-commit mode for the Connection object con so that the transaction will not be automatically committed or rolled back when the methodexecuteBatch is called.

this.con.setAutoCommit(false);

To allow for correct error handling, you should always disable auto-commit mode before beginning a batch update.

The method Statement.addBatch adds a command to the list of commands associated with the Statement object stmt. In this example, these commands are all INSERT INTO statements, each one adding a row consisting of five column values. The values for the columns COF\_NAME and PRICE are the name of the coffee and its price, respectively. The second value in each row is 49 because that is the identification number for the supplier, Superior Coffee. The last two values, the entries for the columnsSALES and TOTAL, all start out being zero because there have been no sales yet. (SALES is the number of pounds of this row's coffee sold in the current week; TOTAL is the total of all the cumulative sales of this coffee.)

The following line sends the four SQL commands that were added to its list of commands to the database to be executed as a batch:

int [] updateCounts = stmt.executeBatch();

Note that stmt uses the method executeBatch to send the batch of insertions, not the method executeUpdate, which sends only one command and returns a single update count. The DBMS executes the commands in the order in which they were added to the list of commands, so it will first add the row of values for Amaretto, then add the row for Hazelnut, then Amaretto decaf, and finally Hazelnut decaf. If all four commands execute successfully, the DBMS will return an update count for each command in the order in which it was executed. The update counts that indicate how many rows were affected by each command are stored in the array updateCounts.

If all four of the commands in the batch are executed successfully, updateCounts will contain four values, all of which are 1 because an insertion affects one row. The list of commands associated with stmt will now be empty because the four commands added previously were sent to the database when stmt called the method executeBatch. You can at any time explicitly empty this list of commands with the method clearBatch.

The Connection.commit method makes the batch of updates to the COFFEES table permanent. This method needs to be called explicitly because the auto-commit mode for this connection was disabled previously.

The following line enables auto-commit mode for the current Connection object.

this.con.setAutoCommit(true);

Now each statement in the example will automatically be committed after it is executed, and it no longer needs to invoke the method commit.

### Performing Parameterized Batch Update

It is also possible to have a parameterized batch update, as shown in the following code fragment, where con is a Connection object:

con.setAutoCommit(false);

PreparedStatement pstmt = con.prepareStatement(

"INSERT INTO COFFEES VALUES( " +

"?, ?, ?, ?, ?)");

pstmt.setString(1, "Amaretto");

pstmt.setInt(2, 49);

pstmt.setFloat(3, 9.99);

pstmt.setInt(4, 0);

pstmt.setInt(5, 0);

pstmt.addBatch();

pstmt.setString(1, "Hazelnut");

pstmt.setInt(2, 49);

pstmt.setFloat(3, 9.99);

pstmt.setInt(4, 0);

pstmt.setInt(5, 0);

pstmt.addBatch();

**// ... and so on for each new**

**// type of coffee**

int [] updateCounts = pstmt.executeBatch();

con.commit();

con.setAutoCommit(true);

### Handling Batch Update Exceptions

You will get a BatchUpdateException when you call the method executeBatch if (1) one of the SQL statements you added to the batch produces a result set (usually a query) or (2) one of the SQL statements in the batch does not execute successfully for some other reason.

You should not add a query (a SELECT statement) to a batch of SQL commands because the method executeBatch, which returns an array of update counts, expects an update count from each SQL statement that executes successfully. This means that only commands that return an update count (commands such as INSERT INTO, UPDATE,DELETE) or that return 0 (such as CREATE TABLE, DROP TABLE, ALTER TABLE) can be successfully executed as a batch with the executeBatch method.

A BatchUpdateException contains an array of update counts that is similar to the array returned by the method executeBatch. In both cases, the update counts are in the same order as the commands that produced them. This tells you how many commands in the batch executed successfully and which ones they are. For example, if five commands executed successfully, the array will contain five numbers: the first one being the update count for the first command, the second one being the update count for the second command, and so on.

BatchUpdateException is derived from SQLException. This means that you can use all of the methods available to an SQLException object with it. The following method, [JDBCTutorialUtilities.printBatchUpdateException](http://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html) prints all of the SQLException information plus the update counts contained in aBatchUpdateException object. Because BatchUpdateException.getUpdateCounts returns an array of int, the code uses a for loop to print each of the update counts:

public static void printBatchUpdateException(BatchUpdateException b) {

System.err.println("----BatchUpdateException----");

System.err.println("SQLState: " + b.getSQLState());

System.err.println("Message: " + b.getMessage());

System.err.println("Vendor: " + b.getErrorCode());

System.err.print("Update counts: ");

int [] updateCounts = b.getUpdateCounts();

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

System.err.print(updateCounts[i] + " ");

}

}

## Inserting Rows in ResultSet Objects

**Note**: Not all JDBC drivers support inserting new rows with the ResultSet interface. If you attempt to insert a new row and your JDBC driver database does not support this feature, a SQLFeatureNotSupportedException exception is thrown.

The following method, [CoffeesTable.insertRow](http://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html), inserts a row into the COFFEES through a ResultSet object:

public void insertRow(String coffeeName, int supplierID,

float price, int sales, int total)

throws SQLException {

Statement stmt = null;

try {

stmt = con.createStatement(

ResultSet.TYPE\_SCROLL\_SENSITIVE

ResultSet.CONCUR\_UPDATABLE);

ResultSet uprs = stmt.executeQuery(

"SELECT \* FROM " + dbName +

".COFFEES");

uprs.moveToInsertRow();

uprs.updateString("COF\_NAME", coffeeName);

uprs.updateInt("SUP\_ID", supplierID);

uprs.updateFloat("PRICE", price);

uprs.updateInt("SALES", sales);

uprs.updateInt("TOTAL", total);

uprs.insertRow();

uprs.beforeFirst();

} catch (SQLException e ) {

JDBCTutorialUtilities.printSQLException(e);

} finally {

if (stmt != null) { stmt.close(); }

}

}

This example calls the Connection.createStatement method with two arguments, ResultSet.TYPE\_SCROLL\_SENSITIVE and ResultSet.CONCUR\_UPDATABLE. The first value enables the cursor of the ResultSet object to be moved both forward and backward. The second value, ResultSet.CONCUR\_UPDATABLE, is required if you want to insert rows into a ResultSet object; it specifies that it can be updatable.

The same stipulations for using strings in getter methods also apply to updater methods.

The method ResultSet.moveToInsertRow moves the cursor to the insert row. The insert row is a special row associated with an updatable result set. It is essentially a buffer where a new row can be constructed by calling the updater methods prior to inserting the row into the result set. For example, this method calls the methodResultSet.updateString to update the insert row's COF\_NAME column to Kona.

The method ResultSet.insertRow inserts the contents of the insert row into the ResultSet object and into the database.

**Note**: After inserting a row with the ResultSet.insertRow, you should move the cursor to a row other than the insert row. For example, this example moves it to before the first row in the result set with the method ResultSet.beforeFirst. Unexpected results can occur if another part of your application uses the same result set and the cursor is still pointing to the insert row.

# JDBC - Batch Processing

Advertisements

[Previous Page](http://www.tutorialspoint.com/jdbc/jdbc-exceptions.htm)

[Next Page](http://www.tutorialspoint.com/jdbc/jdbc-stored-procedure.htm)

Batch Processing allows you to group related SQL statements into a batch and submit them with one call to the database.

When you send several SQL statements to the database at once, you reduce the amount of communication overhead, thereby improving performance.

* JDBC drivers are not required to support this feature. You should use the*DatabaseMetaData.supportsBatchUpdates()* method to determine if the target database supports batch update processing. The method returns true if your JDBC driver supports this feature.
* The **addBatch()** method of *Statement, PreparedStatement,* and *CallableStatement*is used to add individual statements to the batch. The **executeBatch()** is used to start the execution of all the statements grouped together.
* The **executeBatch()** returns an array of integers, and each element of the array represents the update count for the respective update statement.
* Just as you can add statements to a batch for processing, you can remove them with the **clearBatch()** method. This method removes all the statements you added with the addBatch() method. However, you cannot selectively choose which statement to remove.

## Batching with Statement Object

Here is a typical sequence of steps to use Batch Processing with Statement Object −

* Create a Statement object using either *createStatement()* methods.
* Set auto-commit to false using *setAutoCommit()*.
* Add as many as SQL statements you like into batch using *addBatch()* method on created statement object.
* Execute all the SQL statements using *executeBatch()* method on created statement object.
* Finally, commit all the changes using *commit()* method.

### Example

The following code snippet provides an example of a batch update using Statement object −

// Create statement object

Statement stmt = conn.createStatement();

// Set auto-commit to false

conn.setAutoCommit(false);

// Create SQL statement

String SQL = "INSERT INTO Employees (id, first, last, age) " +

"VALUES(200,'Zia', 'Ali', 30)";

// Add above SQL statement in the batch.

stmt.addBatch(SQL);

// Create one more SQL statement

String SQL = "INSERT INTO Employees (id, first, last, age) " +

"VALUES(201,'Raj', 'Kumar', 35)";

// Add above SQL statement in the batch.

stmt.addBatch(SQL);

// Create one more SQL statement

String SQL = "UPDATE Employees SET age = 35 " +

"WHERE id = 100";

// Add above SQL statement in the batch.

stmt.addBatch(SQL);

// Create an int[] to hold returned values

int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes

conn.commit();

For a better understanding, let us study the [**Batching - Example Code**](http://www.tutorialspoint.com/jdbc/statement-batching-example.htm).

## Batching with PrepareStatement Object

Here is a typical sequence of steps to use Batch Processing with PrepareStatement Object −

1. Create SQL statements with placeholders.
2. Create PrepareStatement object using either *prepareStatement()* methods.
3. Set auto-commit to false using *setAutoCommit()*.
4. Add as many as SQL statements you like into batch using *addBatch()* method on created statement object.
5. Execute all the SQL statements using *executeBatch()* method on created statement object.
6. Finally, commit all the changes using *commit()* method.

The following code snippet provides an example of a batch update using PrepareStatement object −

// Create SQL statement

String SQL = "INSERT INTO Employees (id, first, last, age) " +

"VALUES(?, ?, ?, ?)";

// Create PrepareStatement object

PreparedStatemen pstmt = conn.prepareStatement(SQL);

//Set auto-commit to false

conn.setAutoCommit(false);

// Set the variables

pstmt.setInt( 1, 400 );

pstmt.setString( 2, "Pappu" );

pstmt.setString( 3, "Singh" );

pstmt.setInt( 4, 33 );

// Add it to the batch

pstmt.addBatch();

// Set the variables

pstmt.setInt( 1, 401 );

pstmt.setString( 2, "Pawan" );

pstmt.setString( 3, "Singh" );

pstmt.setInt( 4, 31 );

// Add it to the batch

pstmt.addBatch();

//add more batches

.

.

.

.

//Create an int[] to hold returned values

int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes

conn.commit();

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **What is DatabaseMetaData? Write an example code.**   |  | | --- | | **Description:** | | DatabaseMetaData is used to know which type of driver we are using and whether is it compatable or JDBC complaint or not. It is used to know all details about database provider as well. |  |  | | --- | | **Code:** | | package com.java2novice.jdbc;  import java.sql.Connection;  import java.sql.DatabaseMetaData;  import java.sql.DriverManager;  import java.sql.SQLException;  public class MyDatabaseMetadata {  public static void main(String a[]){    Connection con = null;  try {  Class.forName("oracle.jdbc.driver.OracleDriver");  con = DriverManager.  getConnection("jdbc:oracle:thin:@<hostname>:<port num>:<DB name>"  ,"user","password");  DatabaseMetaData dm = con.getMetaData();  System.out.println(dm.getDriverVersion());  System.out.println(dm.getDriverName());  System.out.println(dm.getDatabaseProductName());  System.out.println(dm.getDatabaseProductVersion());  } catch (SQLException e) {  // TODO Auto-generated catch block  e.printStackTrace();  } finally{  if(con != null){}  try {  con.close();  } catch (SQLException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  }  }  } | |
| |  | | --- | | [<< Previous Program](http://www.java2novice.com/jdbc/read-image/) | [Next Program >>](http://www.java2novice.com/jdbc/connection-with-properties/) | |
|  |
| [blog comments powered by Disqus](http://disqus.com/) |
| **List Of All JDBC Programs:**   1. [How to create JDBC Connection?](http://www.java2novice.com/jdbc/create-connection/) 2. [What is Statement & ResultSet in JDBC?](http://www.java2novice.com/jdbc/statement-resultset/) 3. [How to execute and read select queries using JDBC?](http://www.java2novice.com/jdbc/read-select/) 4. [How to update a record in the database using JDBC?](http://www.java2novice.com/jdbc/update-record/) 5. [How to execute any type of query in JDBC?](http://www.java2novice.com/jdbc/execute-method/) 6. [What are the types of JDBC Statements available?](http://www.java2novice.com/jdbc/statement-types/) 7. [Write an example code for JDBC prepared statement.](http://www.java2novice.com/jdbc/prepared-statement/) 8. [Write an example for JDBC prepared statement with ResultSet.](http://www.java2novice.com/jdbc/prepared-statement-resultset/) 9. [How to get primary key value (auto-generated keys) from inserted queries using JDBC?](http://www.java2novice.com/jdbc/auto-generated-keys/) 10. [Write a simple program for CallableStatement statement to execute stored procedure.](http://www.java2novice.com/jdbc/simple-callable-statement/) 11. [Write a program for CallableStatement statement with stored procedure returns OUT parameters.](http://www.java2novice.com/jdbc/out-callable-statement/) 12. [Write a program for CallableStatement statement with batch execution.](http://www.java2novice.com/jdbc/batch-callable-statement/) 13. [Write a program to execute SQL function using CallableStatement.](http://www.java2novice.com/jdbc/function-callable-statement/) 14. [Write a program to execute SQL cursors using CallableStatement.](http://www.java2novice.com/jdbc/cursor-callable-statement/) 15. [How to get column properties from ResultSet using ResultSetMetaData?](http://www.java2novice.com/jdbc/result-set-metadata/) 16. [Write an example for batch update using Statement.](http://www.java2novice.com/jdbc/statement-batch-update/) 17. [Write an example for batch update using PreparedStatement.](http://www.java2novice.com/jdbc/prepared-statement-batch-update/) 18. [What are the types of ResultSets in JDBC?](http://www.java2novice.com/jdbc/types-of-resultset/) 19. [Write an example for scrollable result set with read only mode.](http://www.java2novice.com/jdbc/scroll-read-only/) 20. [Write an example for updatable result set.](http://www.java2novice.com/jdbc/updatable-result-set/) 21. [How to insert an image into database table? or Write an example for inserting BLOB into table.](http://www.java2novice.com/jdbc/insert-image/) 22. [How to read an image from database table? or Write an example for reading BLOB from table.](http://www.java2novice.com/jdbc/read-image/) 23. [What is DatabaseMetaData? Write an example code.](http://www.java2novice.com/jdbc/database-metadata/) 24. [How to get JDBC Connection object using properties file?](http://www.java2novice.com/jdbc/connection-with-properties/) |

**Chapter 21:Fundamental Classes**

In this tutorial you will learn the fundamental wrapper classes. You will also learn about few utility classes like Math class,Calendar and scanner classes.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | |  | | --- | | **0. Use fundamental classes from java.lang package**    :    [**Link**](http://www.ii.uib.no/~khalid/pgjc2e/JC2_Ch10.pdf) | | | |  | | --- | | **0. Object, Class**    :    [**Link**](http://docs.oracle.com/javase/tutorial/java/IandI/objectclass.html) |   **Object as a Superclass**  The [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) class, in the java.lang package, sits at the top of the class hierarchy tree. Every class is a descendant, direct or indirect, of the Object class. Every class you use or write inherits the instance methods of Object. You need not use any of these methods, but, if you choose to do so, you may need to override them with code that is specific to your class. The methods inherited from Object that are discussed in this section are:   * protected Object clone() throws CloneNotSupportedException       Creates and returns a copy of this object. * public boolean equals(Object obj)       Indicates whether some other object is "equal to" this one. * protected void finalize() throws Throwable       Called by the garbage collector on an object when garbage       collection determines that there are no more references to the object * public final Class getClass()       Returns the runtime class of an object. * public int hashCode()       Returns a hash code value for the object. * public String toString()       Returns a string representation of the object.   The notify, notifyAll, and wait methods of Object all play a part in synchronizing the activities of independently running threads in a program, which is discussed in a later lesson and won't be covered here. There are five of these methods:   * public final void notify() * public final void notifyAll() * public final void wait() * public final void wait(long timeout) * public final void wait(long timeout, int nanos)   **Note:** There are some subtle aspects to a number of these methods, especially the clone method.  **The clone() Method**  If a class, or one of its superclasses, implements the Cloneable interface, you can use the clone() method to create a copy from an existing object. To create a clone, you write:  *aCloneableObject*.clone();  Object's implementation of this method checks to see whether the object on which clone() was invoked implements the Cloneable interface. If the object does not, the method throws a CloneNotSupportedException exception. Exception handling will be covered in a later lesson. For the moment, you need to know that clone() must be declared as  protected Object clone() throws CloneNotSupportedException  or:  public Object clone() throws CloneNotSupportedException  if you are going to write a clone() method to override the one in Object.  If the object on which clone() was invoked does implement the Cloneable interface, Object's implementation of the clone() method creates an object of the same class as the original object and initializes the new object's member variables to have the same values as the original object's corresponding member variables.  The simplest way to make your class cloneable is to add implements Cloneable to your class's declaration. then your objects can invoke the clone() method.  For some classes, the default behavior of Object's clone() method works just fine. If, however, an object contains a reference to an external object, say ObjExternal, you may need to override clone() to get correct behavior. Otherwise, a change in ObjExternal made by one object will be visible in its clone also. This means that the original object and its clone are not independent—to decouple them, you must override clone() so that it clones the object *and* ObjExternal. Then the original object referencesObjExternal and the clone references a clone of ObjExternal, so that the object and its clone are truly independent.  **The equals() Method**  The equals() method compares two objects for equality and returns true if they are equal. The equals() method provided in the Object class uses the identity operator (==) to determine whether two objects are equal. For primitive data types, this gives the correct result. For objects, however, it does not. The equals() method provided byObject tests whether the object *references* are equal—that is, if the objects compared are the exact same object.  To test whether two objects are equal in the sense of *equivalency* (containing the same information), you must override the equals() method. Here is an example of a Bookclass that overrides equals():  public class Book {  ...  public boolean equals(Object obj) {  if (obj instanceof Book)  return ISBN.equals((Book)obj.getISBN());  else  return false;  }  }  Consider this code that tests two instances of the Book class for equality:  // Swing Tutorial, 2nd edition  Book firstBook = new Book("0201914670");  Book secondBook = new Book("0201914670");  if (firstBook.equals(secondBook)) {  System.out.println("objects are equal");  } else {  System.out.println("objects are not equal");  }  This program displays objects are equal even though firstBook and secondBook reference two distinct objects. They are considered equal because the objects compared contain the same ISBN number.  You should always override the equals() method if the identity operator is not appropriate for your class.  **Note:** If you override equals(), you must override hashCode() as well.  **The finalize() Method**  The Object class provides a callback method, finalize(), that *may be* invoked on an object when it becomes garbage. Object's implementation of finalize() does nothing—you can override finalize() to do cleanup, such as freeing resources.  The finalize() method *may be* called automatically by the system, but when it is called, or even if it is called, is uncertain. Therefore, you should not rely on this method to do your cleanup for you. For example, if you don't close file descriptors in your code after performing I/O and you expect finalize() to close them for you, you may run out of file descriptors.  **The getClass() Method**  You cannot override getClass.  The getClass() method returns a Class object, which has methods you can use to get information about the class, such as its name (getSimpleName()), its superclass (getSuperclass()), and the interfaces it implements (getInterfaces()). For example, the following method gets and displays the class name of an object:  void printClassName(Object obj) {  System.out.println("The object's" + " class is " +  obj.getClass().getSimpleName());  }  The [Class](https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html) class, in the java.lang package, has a large number of methods (more than 50). For example, you can test to see if the class is an annotation (isAnnotation()), an interface (isInterface()), or an enumeration (isEnum()). You can see what the object's fields are (getFields()) or what its methods are (getMethods()), and so on.  **The hashCode() Method**  The value returned by hashCode() is the object's hash code, which is the object's memory address in hexadecimal.  By definition, if two objects are equal, their hash code *must also* be equal. If you override the equals() method, you change the way two objects are equated and Object's implementation of hashCode() is no longer valid. Therefore, if you override the equals() method, you must also override the hashCode() method as well.  **The toString() Method**  You should always consider overriding the toString() method in your classes.  The Object's toString() method returns a String representation of the object, which is very useful for debugging. The String representation for an object depends entirely on the object, which is why you need to override toString() in your classes.  You can use toString() along with System.out.println() to display a text representation of an object, such as an instance of Book:  System.out.println(firstBook.toString());  which would, for a properly overridden toString() method, print something useful, like this: | | |  | | --- | | **0. Number, Math**    :    [**Link**](http://www.tutorialspoint.com/java/lang/java_lang_number.htm)  [**Link**](http://corejavaassignment.blogspot.in/2009/09/last-post-assignment-on-collection.html) |   Introduction  The **java.lang.Number** class is the superclass of classes BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, and Short.The Subclasses of Number must provide methods to convert the represented numeric value to byte, double, float, int, long, and short.  Class declaration  Following is the declaration for **java.lang.Number** class:  public abstract class Number  extends Object  implements Serializable  Class constructors   |  |  | | --- | --- | | **S.N.** | **Constructor & Description** | | 1 | **Number()**  This is the Single Constructor. |   Class methods   |  |  | | --- | --- | | **S.N.** | **Method & Description** | | 1 | [**byte byteValue()**](http://www.tutorialspoint.com/java/lang/number_bytevalue.htm)  This method returns the value of the specified number as a byte. | | 2 | [**abstract double doubleValue()**](http://www.tutorialspoint.com/java/lang/number_doublevalue.htm)  This method returns the value of the specified number as a double. | | 3 | [**abstract float floatValue()**](http://www.tutorialspoint.com/java/lang/number_floatvalue.htm)  This method returns the value of the specified number as a float. | | 4 | [**abstract int intValue()**](http://www.tutorialspoint.com/java/lang/number_intvalue.htm)  This method returns the value of the specified number as a int. | | 5 | [**abstract long longValue()**](http://www.tutorialspoint.com/java/lang/number_longvalue.htm)  This method returns the value of the specified number as a long. | | 6 | [**short shortValue()**](http://www.tutorialspoint.com/java/lang/number_shortvalue.htm)  This method returns the value of the specified number as a short. |   Methods inherited  This class inherits methods from the following classes:   * java.lang.Object | | |  | | --- | | **0. Calendar**    :    [**Link**](http://www.mkyong.com/java/java-date-and-calendar-examples/)  [**Link**](http://etutorials.org/Misc/programmers+guide+java+certification/Chapter+10.+Fundamental+Classes/Programming+Exercises/) |   Java Date and Calendar examples  By [**mkyong**](http://www.mkyong.com/author/mkyong/) | October 21, 2013 | Updated : January 22, 2015  Calendar  This tutorial shows you how to work with java.util.Date and java.util.Calendar.  1. Java Date Examples  Few examples to work with Date APIs.  **Example 1.1** – Convert Date to String.  SimpleDateFormat sdf = **new** SimpleDateFormat("dd/M/yyyy");  String date = sdf.format(**new** Date());  System.out.println(date); *//15/10/2013*  **Example 1.2** – Convert String to Date.  SimpleDateFormat sdf = **new** SimpleDateFormat("dd-M-yyyy hh:mm:ss");  String dateInString = "31-08-1982 10:20:56";  Date date = sdf.parse(dateInString);  System.out.println(date); *//Tue Aug 31 10:20:56 SGT 1982*  *P.S Refer to this –*[***SimpleDateFormat JavaDoc***](http://docs.oracle.com/javase/6/docs/api/java/text/SimpleDateFormat.html)*for detail date and time patterns.*  **Example 1.3** – Get current date time  SimpleDateFormat dateFormat = **new** SimpleDateFormat("yyyy/MM/dd HH:mm:ss");  Date date = **new** Date();  System.out.println(dateFormat.format(date)); *//2013/10/15 16:16:39*  **Example 1.4** – Convert Calendar to Date  Calendar calendar = Calendar.getInstance();  Date date = calendar.getTime();  2. Java Calendar Examples  Few examples to work with Calendar APIs.  **Example 2.1** – Get current date time  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy MMM dd HH:mm:ss");  Calendar calendar = **new** GregorianCalendar(2013,0,31);  System.out.println(sdf.format(calendar.getTime()));  Output  2013 Jan 31 00:00:00  **Example 2.2** – Simple Calendar example  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy MMM dd HH:mm:ss");  Calendar calendar = **new** GregorianCalendar(2013,1,28,13,24,56);    **int** year = calendar.get(Calendar.YEAR);  **int** month = calendar.get(Calendar.MONTH); *// Jan = 0, dec = 11*  **int** dayOfMonth = calendar.get(Calendar.DAY\_OF\_MONTH);  **int** dayOfWeek = calendar.get(Calendar.DAY\_OF\_WEEK);  **int** weekOfYear = calendar.get(Calendar.WEEK\_OF\_YEAR);  **int** weekOfMonth= calendar.get(Calendar.WEEK\_OF\_MONTH);    **int** hour = calendar.get(Calendar.HOUR); *// 12 hour clock*  **int** hourOfDay = calendar.get(Calendar.HOUR\_OF\_DAY); *// 24 hour clock*  **int** minute = calendar.get(Calendar.MINUTE);  **int** second = calendar.get(Calendar.SECOND);  **int** millisecond= calendar.get(Calendar.MILLISECOND);    System.out.println(sdf.format(calendar.getTime()));    System.out.println("year **\t\t**: " + year);  System.out.println("month **\t\t**: " + month);  System.out.println("dayOfMonth **\t**: " + dayOfMonth);  System.out.println("dayOfWeek **\t**: " + dayOfWeek);  System.out.println("weekOfYear **\t**: " + weekOfYear);  System.out.println("weekOfMonth **\t**: " + weekOfMonth);    System.out.println("hour **\t\t**: " + hour);  System.out.println("hourOfDay **\t**: " + hourOfDay);  System.out.println("minute **\t\t**: " + minute);  System.out.println("second **\t\t**: " + second);  System.out.println("millisecond **\t**: " + millisecond);  Output  2013 Feb 28 13:24:56  year : 2013  month : 1  dayOfMonth : 28  dayOfWeek : 5  weekOfYear : 9  weekOfMonth : 5  hour : 1  hourOfDay : 13  minute : 24  second : 56  millisecond : 0  **Example 2.3** – Set a date manually.  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy MMM dd HH:mm:ss");    Calendar calendar = **new** GregorianCalendar(2013,1,28,13,24,56);  System.out.println("#1. " + sdf.format(calendar.getTime()));    *//update a date*  calendar.set(Calendar.YEAR, 2014);  calendar.set(Calendar.MONTH, 11);  calendar.set(Calendar.MINUTE, 33);    System.out.println("#2. " + sdf.format(calendar.getTime()));  Output  *#1. 2013 Feb 28 13:24:56*  *#2. 2014 Dec 28 13:33:56*  **Example 2.4**– Add or subtract from a date.  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy MMM dd");    Calendar calendar = **new** GregorianCalendar(2013,10,28);  System.out.println("Date : " + sdf.format(calendar.getTime()));    *//add one month*  calendar.add(Calendar.MONTH, 1);  System.out.println("Date : " + sdf.format(calendar.getTime()));    *//subtract 10 days*  calendar.add(Calendar.DAY\_OF\_MONTH, -10);  System.out.println("Date : " + sdf.format(calendar.getTime()));  Output  Date : 2013 Nov 28  Date : 2013 Dec 28  Date : 2013 Dec 18  **Example 2.5**– Convert Date to Calendar.  SimpleDateFormat sdf = **new** SimpleDateFormat("dd-M-yyyy hh:mm:ss");  String dateInString = "22-01-2015 10:20:56";  Date date = sdf.parse(dateInString);    Calendar calendar = Calendar.getInstance();  calendar.setTime(date); | | |  | | --- | | **0. GregorianCalendar**    :    [**Link**](http://www.tutorialspoint.com/java/util/gregoriancalendar_add.htm) |   Java.util.GregorianCalendar.add() Method  Advertisements  [Previous Page](http://www.tutorialspoint.com/java/util/java_util_gregoriancalendar.htm)  [Next Page](http://www.tutorialspoint.com/java/util/java_util_gregoriancalendar.htm)  Description  The **java.util.GregorianCalendar.add(int field,int amount)** method adds the specified (signed) amount of time to the given calendar field, based on the calendar's rules.  Declaration  Following is the declaration for **java.util.GregorianCalendar.add()** method  public void add(int field,int amount)  Parameters   * **field** -- the calendar field. * **amount** -- the amount of date or time to be added to the field.   Return Value  This method does not return a value  Exception   * **IllegalArgumentException** -- if field is ZONE\_OFFSET, DST\_OFFSET, or unknown, or if any calendar fields have out-of-range values in non-lenient mode.   Example  The following example shows the usage of java.util.GregorianCalendar.add() method.  package com.tutorialspoint;  import java.util.\*;  public class GregorianCalendarDemo {  public static void main(String[] args) {  // create a new calendar  GregorianCalendar cal =  (GregorianCalendar) GregorianCalendar.getInstance();  // print the current date and time  System.out.println("" + cal.getTime());  // add 2 months  cal.add((GregorianCalendar.MONTH), 2);  // print the modified date and time  System.out.println("" + cal.getTime());  // add 2 years  cal.add((GregorianCalendar.YEAR), 2);  // print the modified date and time  System.out.println("" + cal.getTime());  }  }  Let us compile and run the above program, this will produce the following result:  Fri May 18 02:25:26 EEST 2012  Wed Jul 18 02:25:26 EEST 2012  Fri Jul 18 02:25:26 EEST 2014 | | |  | | --- | | **0. Scanner**    :    [**Link**](http://www.java-made-easy.com/java-scanner.html) |   **Java Input - Using Java Scanner**  **Introduction**  **Note:** If you're looking for Java Scanner help, [click here](http://www.java-made-easy.com/java-scanner-help.html). The page contains some common questions about them and a question form where you can ask your own questions about Scanners in Java. Otherwise, for a Scanner example scroll down near the bottom of the page. Otherwise, simply follow along with the tutorial. The screen shots throughout are of the [Eclipse IDE](http://www.java-made-easy.com/install-java.html). If you don't have Eclipse, I highly recommend downloading it!  **Want to get more tips and more personalized help on Java Scanner and other Java topics?** Need help with common problems found on quizzes, tests, or other exams? [**Click here**](http://www.java-made-easy.com/java-newsletter.html) to sign up for the **Fresh Cup Of Java Newsletter**, a monthly e-zine full of sample programs, general tips, and other Java related materials, free of charge!  A Java Scanner is the fastest, easiest way to get input from a user in Java.  By this point you should be able display some sort of output onto the screen. You should also be able to store data inside of variables. Both of these concepts are great, but what's the point of a program that has no interaction with a user?  For example, what good is a computer game if you can't control any of it? What we need is input, and Java has plenty of ways of accepting input. However, we'll be using a Java Scanner for this tutorial. We're also going to deal with the simplest kind, text input. This means that we'll be inputting from the keyboard.  **Note:** Looking for Java Scanner help? Check out these [frequently asked Scanner questions!](http://www.java-made-easy.com/java-scanner-help.html)  **Getting Input**  So, first thing's first, we're going to create a new class (a new Java file). If you already have a Java project in Eclipse, then all you have to do is right click on the src folder and hit new class. If you do not have a Java project, you will need to create one. If you don't know what I'm talking about, you really need to read [this tutorial](http://www.java-made-easy.com/java-hello-world.html). It will teach you everything you need to know to get started with writing a program.  Call your new class Inputs. Remember, the class must start with a capital letter. Next, make sure to checkmark the box that adds public static void main(String[] args). If you do not, that's ok, but you'll have to add it in manually. When you've entered the name for your new class, hit finish and you should see this:  http://www.java-made-easy.com/image-files/xinputs.jpg.pagespeed.ic.2vtAzrxcCQ.jpg  There may also be some comments in there that are green and blue (mine did not have those), and you can delete those if you want or leave them alone.  Now we're going to be doing all of our work inside of main. To make sure you're up and running, add a print line inside of the main as shown below (remember, this means in between the opening and closing curly brackets that belong to the main):  http://www.java-made-easy.com/image-files/xprintinputs.jpg.pagespeed.ic.H0C8H4MFY4.jpg  You should know how to do this basic output by now. Make sure that your program runs. You should have seen **Inputs** on the bottom panel of the screen.  Okay, so now onto inputting. First of all you will need to use a Java Scanner that will get this input for you. It acts like a variable but it's not one of the basic types that were talked about in the [previous tutorial](http://www.java-made-easy.com/java-variables.html). Add this line into the main:  It is **EXTREMELY** important that you get the capitalization correct, or this will not work.  You will also notice that Eclipse has underlined Scanner in red, as shown in the next screenshot. That means that Eclipse sees this as an error. That is okay, because the code is actually missing something important.  http://www.java-made-easy.com/image-files/xinputserror.jpg.pagespeed.ic.X9yFzxnz49.jpg  To fix this, right click on the main screen, go down to where it says Source, then select Organize Imports. This will import everything that your program is missing. If a box pops up asking you to choose a type to import, choose java.util.Scanner. This is the correct import for the Java Scanner, as opposed to some other Scanner that might exist. For me it was the first option.  http://www.java-made-easy.com/image-files/xorganizeimports.jpg.pagespeed.ic.TPo0_m_lkq.jpg  You'll notice that the following line was inserted after you organized your imports:  The Java Scanner class is like any class you create, except it was created for you. Since Java already comes with it, it had to be imported.  If you don't want to use the shortcut for importing classes, you can always just type in the import statement manually at the top of the page before the public class line. More on this for a future article though.  What we need to know for this tutorial is that we have a variable called scan, and it is a Scanner. Just like int num = 3; means that num is an int, Scanner scan = new Scanner(System.in); means that scan is a Scanner. It's pretty simple. Don't worry too much about what it equals, it just means that it will be getting our input. Notice how it looks awfully similar to System.out but with an "in" instead.  Just because we have this variable "scan" that will take input does not mean that's all we need to be able to get some input. We have to make it accept some input. To do this, put this line of code right under the line where you create the variable scan:  This will receive the input of the next line of text someone types into the keyboard. It's pretty simple. Now, how can we use the Java Scanner to receive the line of text from a user? Well, we'll need to use a variable. If you're not up to speed on these, you'll need to go take a look at the [previous tutorial](http://www.java-made-easy.com/java-variables.html).  Okay, so now we're going to create a String variable, and we're going to make it be equal to the input we get. Change the line you just wrote to:  Keep in mind that the names of my variables are my preference. You could easily just do:  http://www.java-made-easy.com/image-files/getinput2.jpg  It's up to you to choose what to name your variables, but in general you should try to make them as descriptive as possible while keeping them short.  Okay, so now the input someone types in will be stored in your String variable from the Java Scanner. You can use that variable to now output back the line of text. The program will just echo whatever is typed in. You should be able to output the string on your own, because you should have learned how to already. Go ahead and write the code to do this.  If you did it correctly, you wrote this under the previous line of code:  http://www.java-made-easy.com/image-files/printoutinput.jpg  That will print the input you received using scan.nextLine();. Go ahead and try running the program. When it runs, you'll have to type in the input yourself. To do this, go to the bottom panel where you normally receive your output, and type on the first blank line you see (if stuff is printed there, you need to go down to the first blank line and begin typing). When you hit enter, Java should print out exactly what you typed. Neat stuff.  **Note:** nextLine() will read lines one at a time, including white spaces! If you want to skip all white space, use next() instead. Next() will read up to the first white space and then stop. If you do next() again, it will skip over the white space and continue reading the next set of characters and stop at the next white space, etc.  http://www.java-made-easy.com/image-files/howtoinput.jpg  http://www.java-made-easy.com/image-files/outputofinput.jpg  **scan.nextLine();** will input all the text that was typed up to the point where the user hits the enter key. It's a quick and easy way to get input and to store it into a String.  **Other Inputs**  There are other ways of getting input. We can use scan.nextInt(); to get an integer from the user and store it in an int variable. Like this:  http://www.java-made-easy.com/image-files/intinput.jpg  Now num has the integer that the user typed. If the user types in something other than an integer, the program will **crash**, so you must be careful. There are ways of dealing with these kinds of errors but as with a lot of details, it's beyond the scope of this tutorial. To print this num variable you do as you normally would for any int variable.  **Using Both Input And Output**  Now that we can do both input and output, let's make a little addition program that makes full use of the Java Scanner class. The program will ask the user to type in a number, ask the user to type in a second number, and then display the addition of the two numbers.  You can create a new Java class or you can just delete everything inside of the main method you're working with now(everything in between main's two curly brackets). If you're making a new file, name the class whatever you want, as long as the first letter is capitalized.  To begin, we must ask the user to type in the first number to add. This means you will output a question for the user onto the screen. You know how to do this, so do it.  Next, we will have the user input the number. Remember how to do this? First, we need to create the Scanner variable:  Remember to right click on the main page, select source, and then select organize imports. Next you need to create an int variable to store the first input.  http://www.java-made-easy.com/image-files/intinput2.jpg  Now repeat the process to ask for the second number. Remember to create a new int variable and to call it differently than the other integer variable you created.  Last, do the addition, and output the result. Do you remember how?  I'll skip some lines before spoiling the answer.    http://www.java-made-easy.com/image-files/addexample.jpg  If you do not remember how to do all of this, I suggest rereading the previous tutorials as it is **VITAL** to get these basic concepts. Notice how I did not create two Scanner variables; you only need one to do all inputs.  Also notice the last line of code above. Remember our trick of adding strings together? Well, that's an example of adding a string and a number together. The variable has no quotes because it is a variable. But wait? Isn't num3 an int? Yes, and remember we do not need quotes to print out ints, so this would work anyway. Easy and useful isn't it?  The Java Scanner can do nextDouble() instead of just nextInt(); if you want decimal numbers. It also has a lot of other options that I won't discuss here, but keep that in mind.  Oh, one last thing, **don't try to scan text with nextLine(); AFTER using nextInt() with the same scanner!** It doesn't work well with Java Scanner, and many Java developers opt to just use another Scanner for integers. You can call these scanners scan1 and scan2 if you want.  So there you have it, that's how you get input using a Java Scanner. I strongly suggest playing around with what you learned and try to make your own little program that accepts user input. In the [next tutorial](http://www.java-made-easy.com/if-statement.html) we'll learn how to make decisions with the user's input.  Read more: <http://www.java-made-easy.com/java-scanner.html#ixzz3gVmerjNT> | |  |