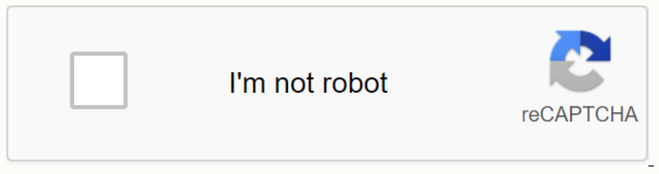

Continue

Classes and objects in java notes pdf

Announcements Reminder: all questions/answers about the curriculum and tasks must go to Piazza. Feel free to stay nameless if you like or ask privately questions that you think are irrelevant to your class. Please just email me or Garrett if you have an administrative problem (for example, attended class, but we don't give credit for class practice). Note about the course's optional textbooks: Yes, books and reading are optional. But optional automatically equal you do not have to make books or do the reading. I'm sensitive that you buy ebooks you don't need (and you'll note that many of the texts are available for free online). But if you need a reference material, or an alternative to presenting the material, or a reference, the texts can help. If you have a late add, you need to contact me to see what you need to do to catch up. SPIRE does not notify me of late adds, and the CS main office just can override the course. If you used SPIRE to add yourself and didn't contact me, I have no idea you added it late. Methods in Java, methods are always related to a class. The method consists of a statement and a body, which is just a series of instructions. Let's take a closer look at the declaration: public static void main(String[] args) { ... } WTF is going on here!?!?! usually the reaction we get is 121 when people first start learning Java. But right now, you probably know enough to understand most of it. Let's work it out inside out. String[] args is a parameter for this method. One or more parameters stick in and look like (and in many respects act as variable declarations). The difference is that their values are provided by the calling method (or, in the case of the main case, the JVM). Here the main get a set of strings that are exactly what pass the command line of the java interpreter, e.g. public class PrintArgs { public static void main(String[] args) { int i = 0; a (String a : args) { System.out.println(i + : + a); } } } } } } > javac PrintArgs.java > java PrintArgs Hello Marc! 0: Hello 1: Marc! The following is the name of the method, which is referred to as camelCased, starting in lowercase letters. Next is the method's return type, or invalid if you don't return anything. Important note: For example, methods that return an invalidation typically print something, such as printing a value or deleting an item from a list or similar. They affect the state or are stateful. Sometimes, but not always, the methods that return the value do not do something (rather mathematical functions). The only way to be sure is to read the documentation (or the method code itself)! But the public API of the method should describe what it does. Then comes one or more methods of modifier: either abstract or one or more static, final and synchronized. static methods are a but not to a specific instance of that class (in other words, not an object). We'll do it. the other modifiers. Finally, legbazis is just a public, protected, or private member access modifier. The objects that can be used to call this method are determined by this modifier. public and private are likely to be familiar to you (all objects and only objects in this class, respectively). protected and no modifier (default access or package access) special meaning will be skipped now. Classes and objects in Java are said to be object-oriented, meaning that language strongly encourages the use of objects as a paradigm for problem solving. What's the object? A collection of related states and behaviors. In Java, this means that objects are usually variables and related methods. The design of objects is a class. The classes are also the place to stick to things that aren't necessarily part of an object. (It falls out of Java design: all parts of a class.) Classes in the JVM (Note that in our discussion of the call stack, this is a simplified version of how you have a mental model of how things work, the implementation of the JVM is somewhat different).) A single instance of the class (note: class! is not a class object!) lives in memory, annotated with its name, by all methods and spacing for each static variable. There are also a few other things: an indicator of the class superclass, and a list of interfaces accomplishes and the like. Objects that serve as instances of the class can refer to these methods and variables here. Instance of objects Objects are defined by a class. But an object doesn't exist until it copies it, that is, it can take a new one to the class as a template for the new object. For example: There is a class called Bus, and we have brought an object of type Bus. The object is named by the variable bus44. Thinking back to our first class, what happens in our memory when that happens? Unlike primitive types that are stored directly in the reserved area of the variable, holding down the object is not actually the object's value. This is a pointer or reference to the place where the object actually lives. This is very important to understand, so I'll tell you again: what Java stores in primitive types of variables and object variables is different, conceptually: The former stores the value well, the latter stores as the value of the memory address of the object. It's weird, but it's true. Let's make an example. What's happening here? String message = new string(Hi!), String what is up = message; A new string is first allocated. Then it initializes on the pile. Then a new variable named a string is created, called a message. Its value is set to the address of the actual String object that you created. Note that this string object implicitly references a string class somewhere else. When you call methods on an object, you use this reference to find the method — a method code is only instance of the class at the same time, all objects in the class share it. Next, another new variable, again the type type named what is being created. The address of the message was visited and then assigned to our solution. Both variables now point or reference the same object, which is a String object that uses Hi! contains data. If you have two variables, the same object is called an alias; this is a common source of program errors. Always carefully consider the value stored in the variable, not just the name of the variable. So this leads to one important thing: == vs .equals(). For primitive types (int and so on), there can be only one option: ==. What good is that? Finds values stored in variables and returns true values when and only if they are the same. x = 3; y = 3; x==y; But what happens if we use == the variables that refer to objects? It's exactly the same! Which may not be what we think. Follow the top to add String hello =new string(Hi!);, and ask if the message == what's going on? Yes, because they refer to the same object. Do you have a message == hello? Not. Although the two string objects represent the same value (Hi!), objects at different addresses are stored in separate objects. Thus, the variables store the values of two different addresses. But often we don't really want to know if there are two variable points on the same objects. Instead, we want to know if the values of the two objects are equivalent. In the case of strings. this means that the same text is kept; You can imagine that with more complex objects, we may need a more complex comparison of the two object instance variables. There is an equals() method that a class can implement to provide a class-specific test of equality. (If not implemented, the object class equals() method, which is used by default ==. So you can write message.equals(hello) to check if the two objects are stored in equivalent strings, rather than two variables storing the same address in a value. Class practice Bus busA = new bus(); Bus bus B = new bus(); Bus busC = busA; busA.setNumber(44); busB.setNumber (44); System.out.println(busA == busB); System.out.println(busA.equals(busB)); busC.setNumber(13); System.out.println(busA == busC); System.out.println(busA.equals(busC)); What are the four line outputs (true/false)? Use objects and methods If the control process takes place within an object and you access a variable, say x, what happens? First, it checks the local scope, turning it inside out, for example: class Bar { int x = 0; private void foo(int j) { for (int i = 0; i<x.length; {x+i] = j; } } x is not declared in the for loop, either in the body of the method or as a parameter, so the next verified location is the object itself where it was found. (The superclasses, etc.) All this is done in translation-time; you can't fail this by looking for a source that translates correctly. In this connection, when methods are called to a the object type is examined and the method is examined in the appropriate class. If not found, the class superclass is checked, and so on. Again, it checked translation time. This is how the default equal method works; object, which by default is a superclass of all classes that would not otherwise declare a superclass. In other words, it inherits the method of superclass. We'll do (slightly) more inheritance, but best practices over the years have shifted to a relatively flat class hierarchy. Typically, you won't see (or use) deep inheritance, with a few exceptions for older code bases and large libraries (such as the Java Platform API). API).