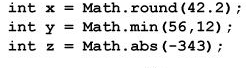
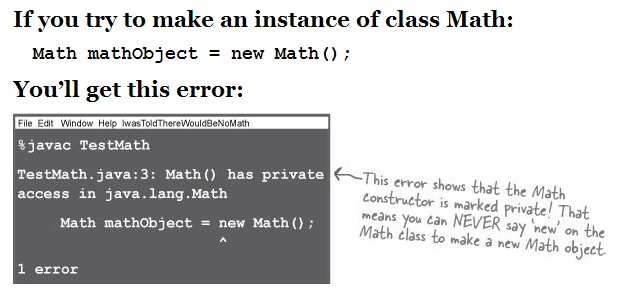
**B”H**

**Chapter 10**

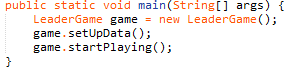
**Numbers and Statics**

* There’s no global anything in Java.
* So what do you do if you have a method whose behavior doesn’t depend on an instance variable value; like the round() method (and many other math methods) in the Math class.
* It would be a waste of heap space to make an instance of class Math simply to run the round() method.
* Methods in the Math class don’t use any instance variable values. And because the methods are ‘static’, you don’t need to have an instance of Math (in fact you can’t). All you need is the Math class.





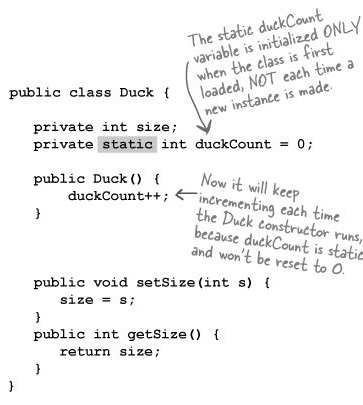
* A static method means “behavior not dependent on an instance variable, so no instance/object is required. Just the class. Hence **int x = Math.round(42.2);**
* Often (although not always), a class with static methods is not meant to be instantiated.
* Math class has a private constructor
* Note how this is different than an abstract class where it can be extended and objects can be created. Unless it’s marked final as well.
* Side note – the Singleton Pattern uses a private constructor yet creates an instance (albeit only one).
* This does not mean that a class with one or more static methods should never be instantiated. In fact, every class you put a main() method in is a class with a static method in it.
* Static methods can’t use non-static (instance) variables nor non-static methods. It won’t compile.
* You can’t even use a non-static method that doesn’t use any instance variables.
* Even though static methods can’t use non-static (instance) variables nor other methods that use non-static (instance) variables – nevertheless they **can** instantiate a class (even its own class as is done often in the **main** method) and then use the non-static (instance) variables or other methods that use non-static (instance) variables from that class. For example the following is valid:



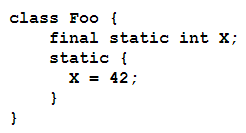
* You can call a static method using a reference variable instead of the class name – but its NOT good practice and makes the code confusing. The compiler just resolves it back to the real class anyway.
* This is legal but NOT good:



* **Static variable**: value is the same for ALL instances of the class
* The static variable is initialized ONLY when the class is first loaded, NOT each time a new instance is made.



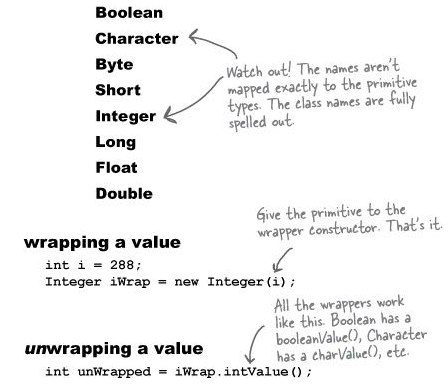
* Static variables in a class are initialized before any object of that class can be created and before any **static method** of the class runs.
* Declaring, but not initializing, a static variable means the static variable will get the default value for that variable type, in exactly the same way that instance variables are given default values when declared.
* **static final** variables are constants. Like Math.PI
* Constant variable names should be in all caps



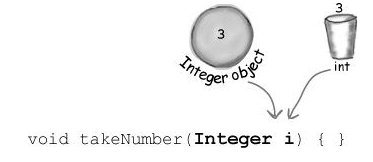
* A static initializer is a block of code that runs only once the first time the class is loaded, before any other code can use the class, so it’s a great place to initialize a complex static variable and especially a complex **static** **final** variable.
* The static initializer block can be used for other code as well that you want to run only once the first time a class is loaded, such as for logging etc.
* See **TestStatic.java** for small example
* You can use the keyword final to modify non-static variables too, including instance variables, local variables, and even method parameters.
* A final variable means you can’t change its value.
* A final method means you can’t override the method.
* A final class means you can’t extend the class (i.e. you can’t make a subclass).
* If the class is final, you obviously don’t need to mark the methods final.

Some Math methods:

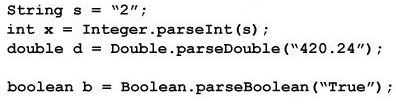
* **Math.random()** Returns a double between 0.0 through (but not including) 1.0.
* **Math.abs()** Returns a double that is the absolute value of the argument. The method is overloaded, so if you pass it an int it returns an int. Pass it a double it returns a double.
* **Math.round()** Returns an int or a long (depending on whether the argument is a float or a double) rounded to the nearest integer value.
* **Math.min()** and **Math.max()** Returns a value that is the minimum/maximum of the two arguments. The method is overloaded to take ints, longs, floats, or doubles.
* Wrapping a primitive: sometimes you want to treat a primitive like an object. For example in ArrayList before Java 5.0



* The autoboxing feature added to Java 5.0 does the conversion from primitive to wrapper object automatically (like used in ArrayList)
* For ArrayList of integers you can’t use ArrayList<int> rather use ArrayList<Integer>.
* If a method takes a wrapper type, you can pass a reference to a wrapper or a primitive of the matching type. And vice-versa.



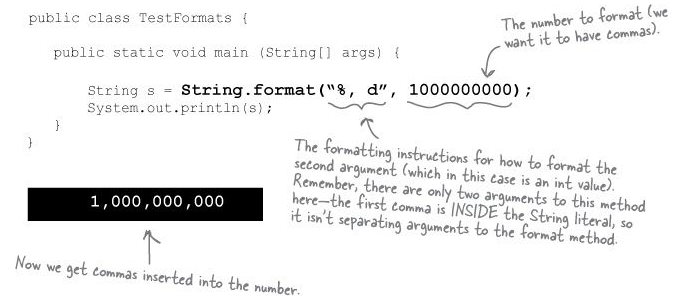
* Same applies for method return values, Boolean expressions, operations on numbers, and assignments
* Wrappers have a bunch of really useful static methods.
* For example:



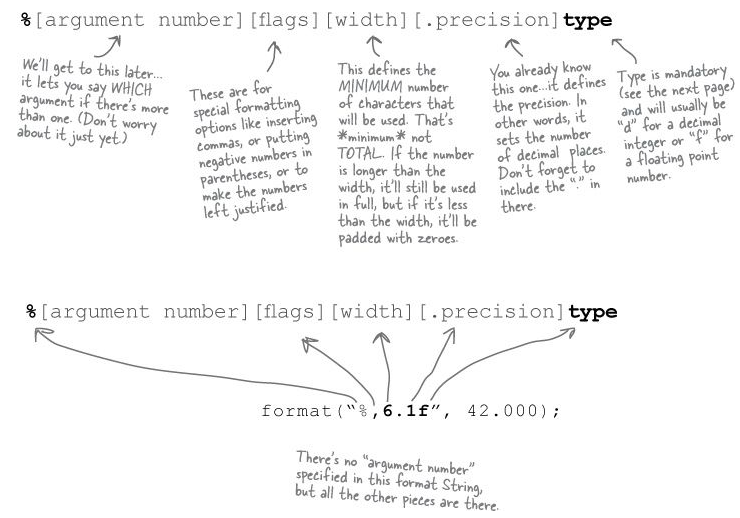
* Every method or constructor that parses a String can throw a NumberFormatException. It’s a runtime exception, so you don’t have to handle or declare it. But you might want to. (We’ll talk about Exceptions in the next chapter.)

**Number formatting**

* Formatting a number to use commas:



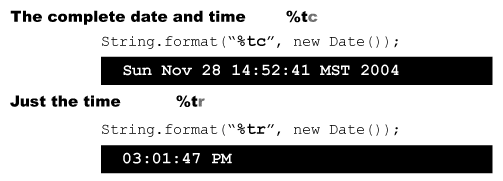
* At the most basic level, formatting consists of two main parts (there is more, but we’ll start with this to keep it cleaner):
  + Formatting instructions.
  + The argument to be formatted.
* There can be more than one argument to be formatted
* The argument type can’t be just anything. It has to be something that can be formatted using the format specifiers in the formatting instructions. For example, if your formatting instructions specify a floating point number, you can’t pass in a Dog or even a String that looks like a floating point number.
* The percent (%) says, “insert argument here” (and format it using these instructions)
* You obviously can’t put just anything after the “%” sign. The syntax for what goes after the percent sign follows very specific rules, and describes how to format the argument that gets inserted at that point in the result (formatted) String.
* Everything after the percent sign up to and including the type indicator (like “d” or “f”) are part of the formatting instructions. After the type indicator, the formatter assumes the next set of characters are meant to be part of the output String.
* A format specifier:
  + Can have up to five different parts (not including the “%”).
  + Everything in brackets [ ] below is optional, so only the percent (%) and the type are required.
  + But the order is also mandatory, so any parts you DO use must go in this order.

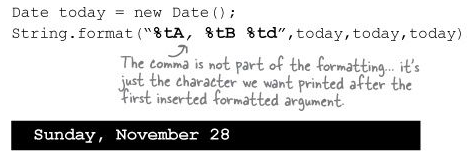


* There are more than a dozen different type modifiers (not including dates and times; they have their own set), but most of the time you’ll probably use %d (decimal) or %f (floating point). And typically you’ll combine %f with a precision indicator to set the number of decimal places you want in your output.
* Using more than one arguement
  + String.format("The rank is %,d out of %,.2f", 18972653, 1897263.877837);
* There are not a bunch of overloaded format() methods to take a different number of possible arguments.
  + In order to support this new formatting (printf-like) API in Java, the language needed another new feature: variable argument lists (called varargs for short).
  + Outside of formatting, you probably won’t use varargs much in a well-designed system (see appendix).

**Date formatting**

* The main difference between number and date formatting is that date formats use a two-character type that starts with “t” (as opposed to the single character “f” or “d”, for example).





* Same as above, but without duplicating the arguments:



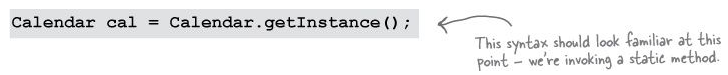
* The angle-bracket **“<”** is just another flag in the specifier that tells the formatter to use the previous argument again.

**Static imports**

* See page 307 for static imports. The basic idea is that whenever you’re using a static class, a static variable, or an enum (more on those later), you can import them, and save yourself some typing.
* But we’re not going to use them because they make your code confusing to read and can create naming conflicts.

**Working with Dates**

* For a time-stamp of “now”, use Date. But for everything else, use Calendar.
* When you want to work with dates, you ask for a Calendar through a static method of the Calendar class and the JVM hands you back an instance of a concrete subclass of Calendar.
* Calendar is actually an abstract class, so you’re always working with a concrete subclass.
* The kind of calendar you get back will be appropriate for your locale. In most of the world, and by default for most versions of Java, you’ll be getting back a java.util.GregorianCalendar instance.



* There are several key concepts you’ll need to understand in order to work with Calendar objects:
  + Fields hold **state**
    - A Calendar object has many fields that are used to represent aspects of its ultimate state, its date and time. For instance, you can get and set a Calendar’s year or month.
  + Dates and Times can be incremented
    - The Calendar class has methods that allow you to add and subtract values from various fields, for example “add one to the month”, or “subtract three years”.
  + Dates and Times can be represented in milliseconds
    - The Calendar class lets you convert your dates into and out of a millisecond representation. (Specifically, the number of milliseconds that have occured since January 1st, 1970.) This allows you to perform precise calculations such as “elapsed time between two times” or “add 63 hours and 23 minutes and 12 seconds to this time”.

