Chapter-10:
Numbers Matter

Upcode Software
Engineer Team

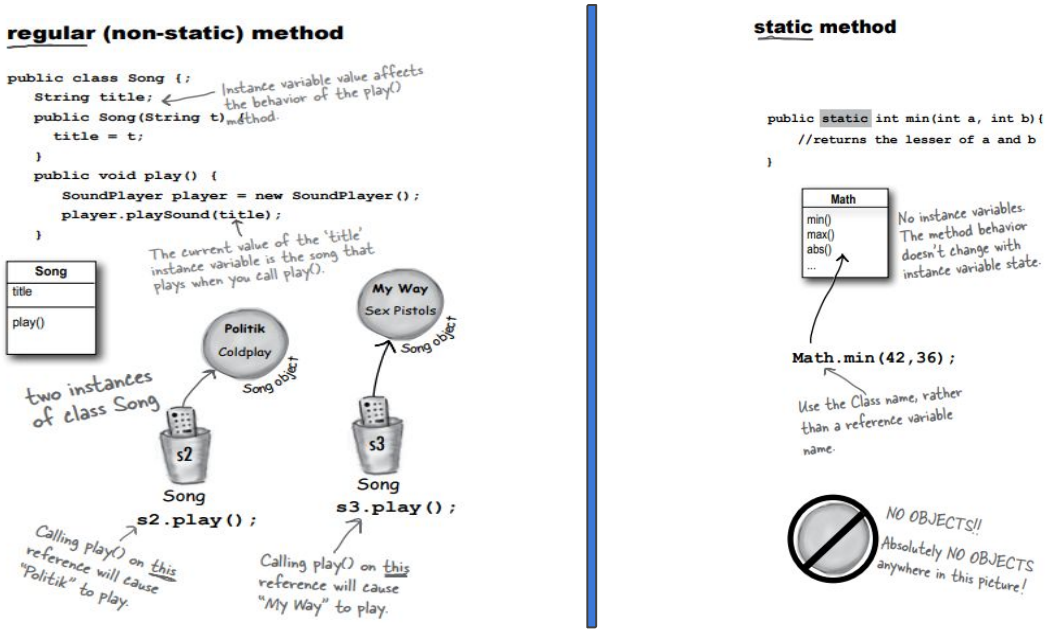
-2023-

CONTENT

1. Static and non-static parameters.
2. Wrapping a primitive.
3. Autoboxing: blurring the line between primitive and object.
4. Autoboxing works almost everywhere.
5. Wrappers have static utility methods too!
6. Primitive number into a String.
7. Number formatting.
8. What about dates ?
9. What about Calendar Api?
10. Static import.

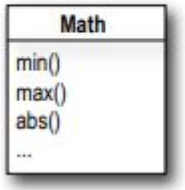
Static and non-static parameters (1/n)

- The difference between regular (non-static) and static methods



Static and non-static parameters (2/n)

Call a static method using a class name




```
Math.min(88,86);
```

Call a non-static method using a reference variable name


```
Song t2 = new Song();
t2.play();
```

Static and non-static parameters (3/n)

- Static methods can’t use non-static (instance) variables!
- If you try to use an instance variable from inside a static method, the compiler thinks, “I don’t know which object’s instance variable you’re talking about!”
- If you have ten Duck objects on the heap, a static method doesn’t know about any of them.

If you try to compile this code:

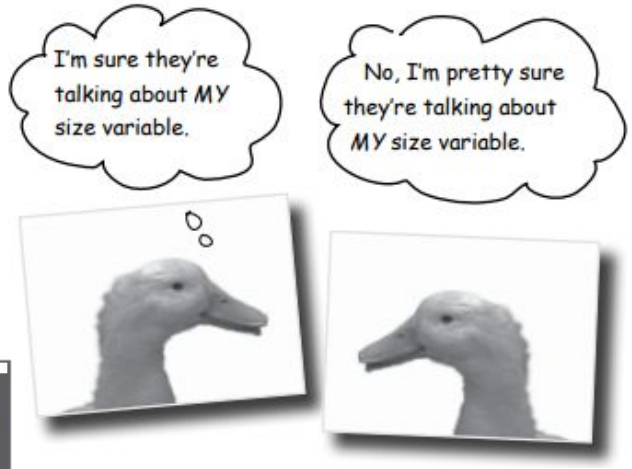
```
public class Duck {

   private int size;

   public static void main (String[] args) {
      System.out.println("Size of duck is " + size);
   }

   public void setSize(int s) {
      size = s;
   }
   public int getSize() {
       return size;
   }
}
```

Which Duck? Whose size?

If there’s a Duck on the heap somewhere, we don’t know about it.

You’ll get this error:

```
File Edit Window Help Quack
% javac Duck.java
Duck.java:6: non-static variable size cannot be referenced from a static context
      System.out.println("Size of duck is " + size);
                                              ^
```



Static and non-static parameters (4/n)

- Static methods can’t use non-static methods, either!

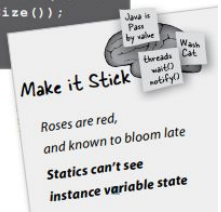
This won’t compile:

```
public class Duck {

   private int size;

   public static void main (String[] args) {
      System.out.println("Size is " + getSize());
   }

   public void setSize(int s) {
      size = s;
   }
   public int getSize() {
       return size;
   }
}
```

Calling getSize() just postpones the inevitable—getSize() uses the size instance variable.

Back to the same problem... whose size?

```
File Edit Window Help Jack-in
% javac Duck.java
Duck.java:6: non-static method
getSize() cannot be referenced
from a static context
      System.out.println("Size
of duck is " + getSize());
                   ^
```



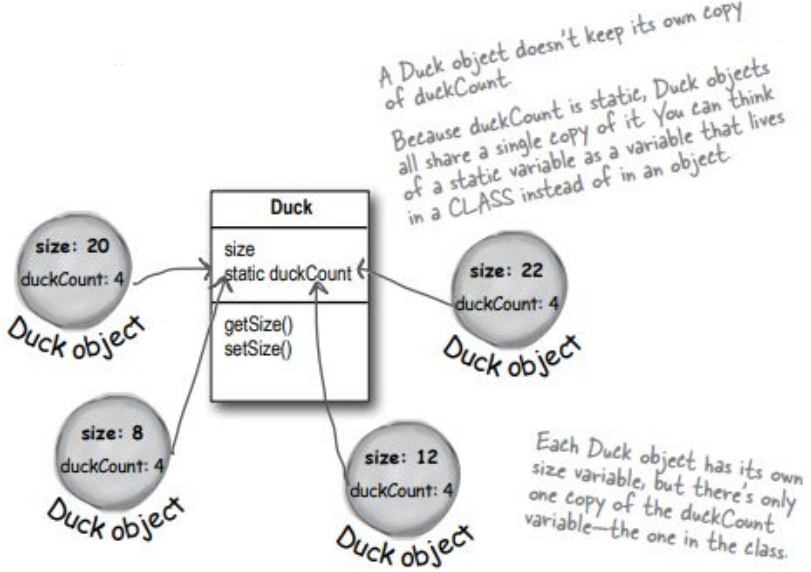
Static and non-static parameters (5/n)

- Static variable: value is the same for ALL instances of the class

```
public class Duck {

    private int size;
    private static int duckCount = 0;

    public Duck() {
        duckCount++;
    }

    public void setSize(int s) {
        size = s;
    }
    public int getSize() {
        return size;
    }
}
```

The static duckCount variable is initialized ONLY when the class is first loaded, NOT each time a new instance is made.

Now it will keep incrementing each time the Duck constructor runs, because duckCount is static and won't be reset to 0.



Static and non-static parameters (6/n)

- Initializing a static variable.
- All static variables in a class are initialized before any object of that class can be created.
- Static variables in a class are initialized before any static method of the class runs.

```
class Player {
    static int playerCount = 0;
    private String name;
    public Player(String n) {
        name = n;
        playerCount++;
    }
}
```

The playerCount is initialized when the class is loaded. We explicitly initialized it to 0, but we don't need to since 0 is the default value for ints. Static variables get default values just like instance variables.

Static and non-static parameters (7/n)

- static final variables are constants.

- The variable is marked final because PI doesn't change (as far as Java is concerned).

```
public static final double PI = 3.141592653589793;
```

A static initializer is a block of code that runs when a class is loaded, before any other code can use the class, so it's a great place to initialize a static final variable.

```
class Foo {
    final static int X;
    static {
      X = 42;
    }
}
```

Static and non-static parameters (8/n)

- Initialize a final static variable:

① At the time you declare it:

```
public class Foo {
   public static final int FOO_X = 25;
}
```

notice the naming convention — static final variables are constants, so the name should be all uppercase, with an underscore separating the words

OR

② In a static initializer:

```
public class Bar {
   public static final double BAR_SIGN;

   static {
       BAR_SIGN = (double) Math.random();
   }
}
```

this code runs as soon as the class is loaded, before any static method is called and even before any static variable can be used.

If you don't give a value to a final variable in one of those two places:

```
public class Bar {
   public static final double BAR_SIGN;
}
```

no initialization!

The compiler will catch it:

```
File Edit Window Help Jack-in
% javac Bar.java
Bar.java:1: variable BAR_SIGN
might not have been initialized
1 error
```

Static and non-static parameters (9/n)

- Final isn't just for static variables...

A final **variable** means you can't **change** its value.

A final **method** means you can't **override** the method.

non-static final variables

```
class Foof {
    final int size = 3;   <- now you can't change size
    final int whuffie;

    Foof() {
       whuffie = 42;   <- now you can't change whuffie
    }

    void doStuff(final int x) {
       // you can't change x
    }

    void doMore() {
       final int z = 7;
       // you can't change z
    }
}
```

final method

```
class Poof {
    final void calcWhuffie() {
       // important things
       // that must never be overridden
    }
}
```

Static and non-static parameters (10/n)

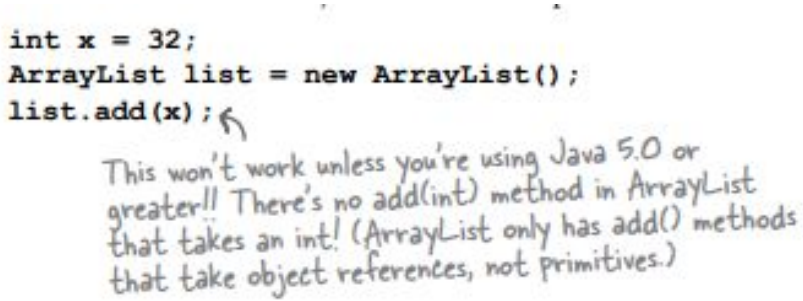
A final **class** means you can't **extend** the class (i.e. you can't make a subclass).

final class

```
final class MyMostPerfectClass {
   // cannot be extended
}
```

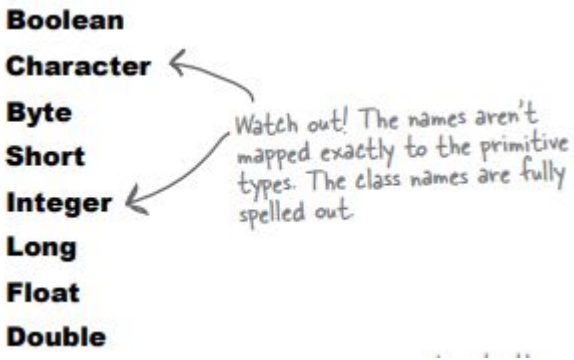
Wrapping a primitive. (1/n)

- Sometimes you want to treat a primitive like an object. For example, in all versions of Java prior to 5.0, you cannot put a primitive directly into a collection like ArrayList or HashMap:



Wrapping a primitive. (2/n)

- There's a wrapper class for every primitive type, and since the wrapper classes are in the java. lang package, you don't need to import them.
- You can recognize wrapper classes because each one is named after the primitive type it wraps, but with the first letter capitalized to follow the class naming convention.



Wrapping a primitive. (3/n)

- When you need to treat a primitive like an object, wrap it. If you're using any version of Java before 5.0, you'll do this when you need to store a primitive value inside a collection like ArrayList or HashMap.

wrapping a value

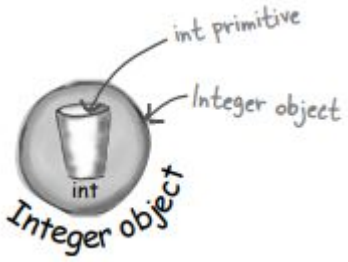
```
int i = 288;
Integer iWrap = new Integer(i);
```

Give the primitive to the wrapper constructor. That's it.

unwrapping a value

```
int unWrapped = iWrap.intValue();
```

All the wrappers work like this. Boolean has a booleanValue(), Character has a charValue(), etc.



Wrapping a primitive. (4/n)

- An ArrayList of primitive ints.

Without autoboxing (Java versions *before* 5.0)

```
public void doNumsOldWay() {
    ArrayList listOfNumbers = new ArrayList();
    listOfNumbers.add(new Integer(3));
    Integer one = (Integer) listOfNumbers.get(0);
    int intOne = one.intValue();
}
```

Make an ArrayList. (Remember, before 5.0 you could not specify the TYPE, so all ArrayLists were lists of Objects.)

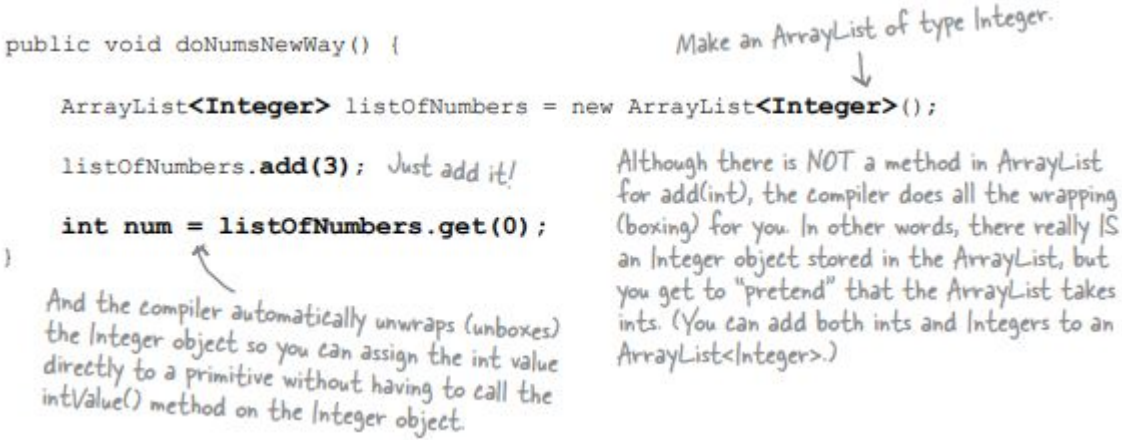
You can't add the primitive '3' to the list, so you have to wrap it in an Integer first.

It comes out as type Object, but you can cast the Object to an Integer.

Finally you can get the primitive out of the Integer.

Blurring the line between primitive and object.

- The autoboxing feature added to Java 5.0 does the conversion from primitive to wrapper object automatically!

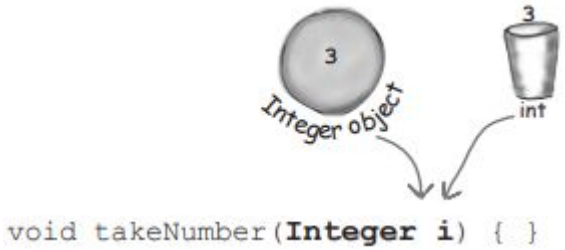


Autoboxing works almost everywhere.

- Autoboxing lets you do more than just the obvious wrapping and unwrapping to use primitives in a collection... it also lets you use either a primitive or its wrapper type virtually anywhere one or the other is expected.
  - **Method arguments.**
  - **Return values.**
  - **Boolean expressions.**
  - **Operations on numbers.**
  - **Assignments.**

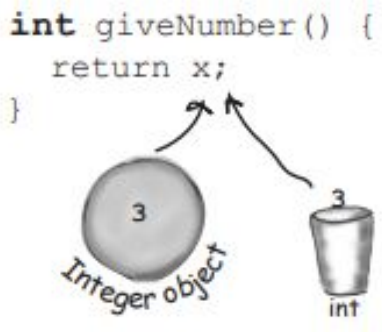
Autoboxing works almost everywhere. (Method)

- **Method arguments.**
  - If a method takes a wrapper type, you can pass a reference to a wrapper or a primitive of the matching type. And of course the reverse is true—if a method takes a primitive, you can pass in either a compatible primitive or a reference to a wrapper of that primitive type.



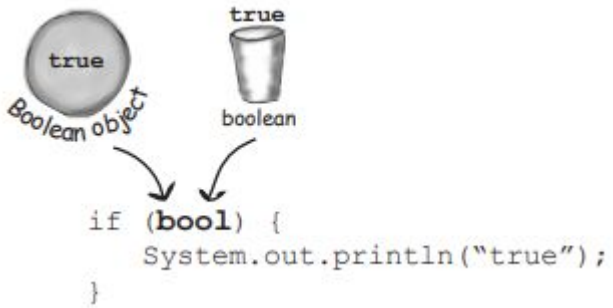
Autoboxing works almost everywhere. (Return)

- **Return values.**
  - If a method declares a primitive return type, you can return either a compatible primitive or a reference to the wrapper of that primitive type. And if a method declares a wrapper return type, you can return either a reference to the wrapper type or a primitive of the matching type.



Autoboxing works almost everywhere. (Boolean)

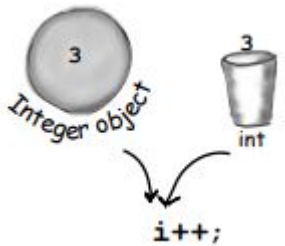
- **Boolean expressions.**
  - Any place a boolean value is expected, you can use either an expression that evaluates to a boolean (4 > 2), or a primitive boolean, or a reference to a Boolean wrapper.



Autoboxing works almost everywhere. (Operations)

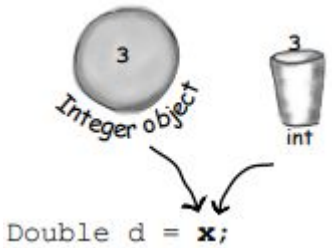
- **Operations on numbers.**
  - This is probably the strangest one—yes, you can now use a wrapper type as an operand in operations where the primitive type is expected. That means you can apply, say, the increment operator against a reference to an Integer object!
  - But don't worry—this is just a compiler trick. The language wasn't modified to make the operators work on objects; the compiler simply converts the object to its primitive type before the operation.

```
Integer i = new Integer(42); i++;
```



Autoboxing works almost everywhere. (Assignments)

- **Assignments.**
  - You can assign either a wrapper or primitive to a variable declared as a matching wrapper or primitive. For example, a primitive int variable can be assigned to an Integer reference variable, and vice-versa—a reference to an Integer object can be assigned to a variable declared as an int primitive.



QUESTION ?

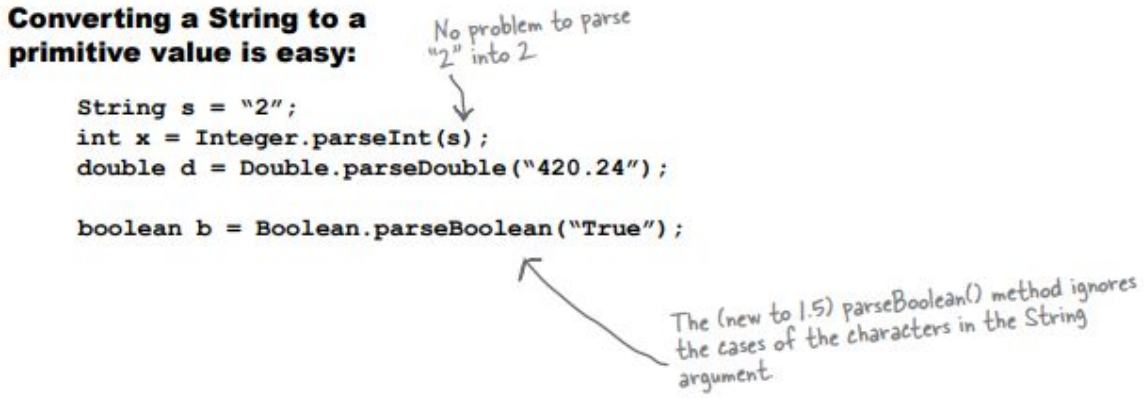
- Will this code compile? Will it run? If it runs, what will it do?

```
public class TestBox {

   Integer i;
   int j;

   public static void main (String[] args) {
      TestBox t = new TestBox();
      t.go();
   }

   public void go() {
      j=i;
      System.out.println(j);
      System.out.println(i);
   }
}
```

```
Exception in thread "main" java.lang.NullPointerException
    at uz.leetcode.math.test.Test.go(Test.java:19)
    at uz.leetcode.math.test.Test.main(Test.java:15)
```

Wrappers have static utility methods too! (1/n)

- Besides acting like a normal class, the wrappers have a bunch of really useful static methods.
- The parse methods take a String and give you back a primitive value.



Wrappers have static utility methods too! (2/n)

- 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 if you try to do this:

```
String t = "two";
int y = Integer.parseInt(t);
```

Uh-oh. This compiles just fine, but at runtime it blows up. Anything that can't be parsed as a number will cause a NumberFormatException

```
File Edit Window Help Clue
% java Wrappers
Exception in thread "main"
java.lang.NumberFormatException: two
at java.lang.Integer.parseInt(Integer.java:409)
at java.lang.Integer.parseInt(Integer.java:458)
at Wrappers.main(Wrappers.java:9)
```

Primitive number into a String.

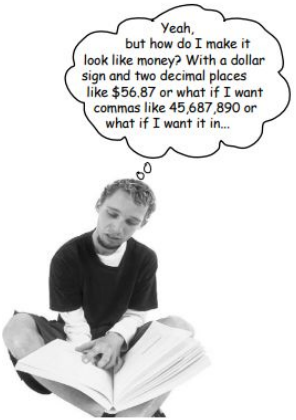
- There are several ways to turn a number into a String.
- The easiest is to simply concatenate the number to an existing String

```
double d = 42.5;
String doubleString = "" + d;
```

Remember the '+' operator is overloaded in Java (the only overloaded operator) as a String concatenator. Anything added to a String becomes Stringified.

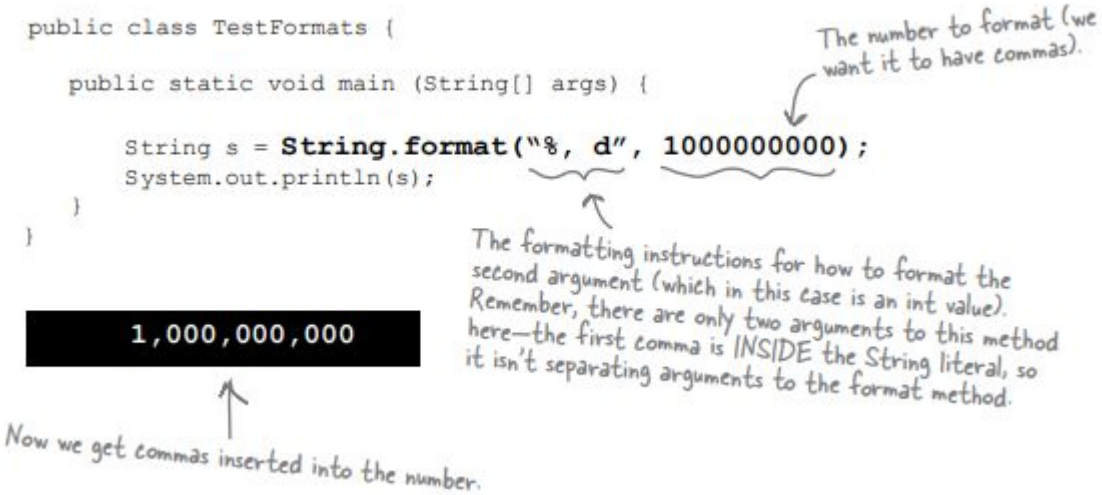
```
double d = 42.5;
String doubleString = Double.toString(d);
```

Another way to do it using a static method in class Double.



Number formatting. (1/n)

- Formatting a number to use commas.



Number formatting. (2/n)

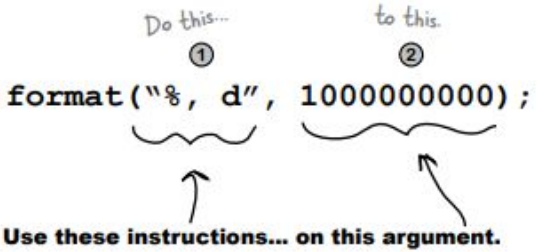
- Formatting deconstructed…
  - At the most basic level, formatting consists of two main parts (there is more, but we'll start with this to keep it cleaner):

① Formatting instructions

You use special format specifiers that describe how the argument should be formatted.

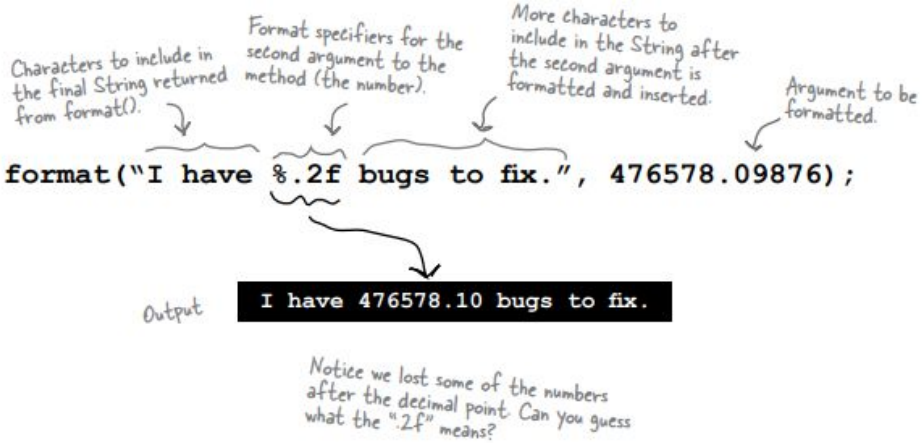
② The argument to be formatted.

Although there can be more than one argument, we'll start with just one. 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.



Number formatting. (3/n)

- The percent (%) says, “insert argument here” (and format it using these instructions).



- The “%” sign tells the formatter to insert the other method argument (the second argument to format(), the number) here, AND format it using the “.2f” characters after the percent sign

Number formatting. (4/n)

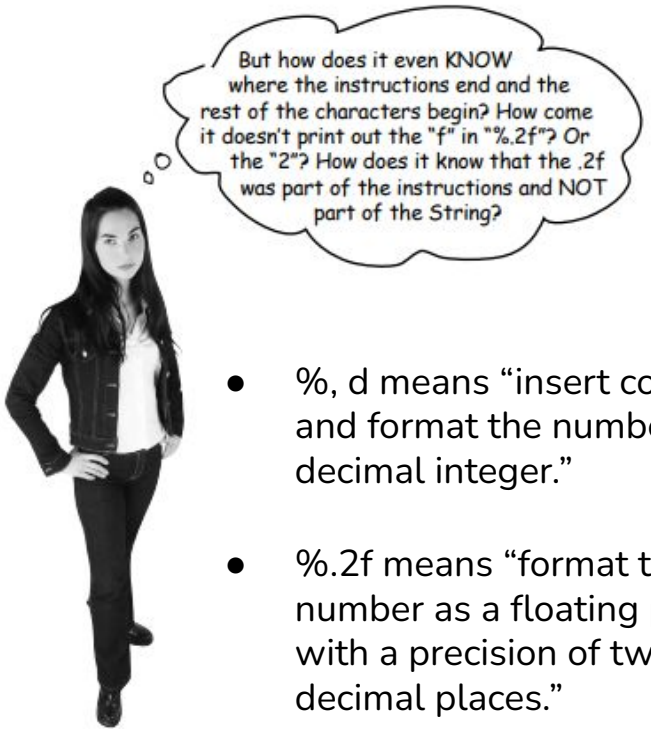
- Adding a comma.

```
format("I have %,.2f bugs to fix.", 476578.09876);
```

```
I have 476,578.10 bugs to fix.
```

By changing the format instructions from "%.2f" to %,.2f", we got a comma in the formatted number.

Number formatting. (5/n)

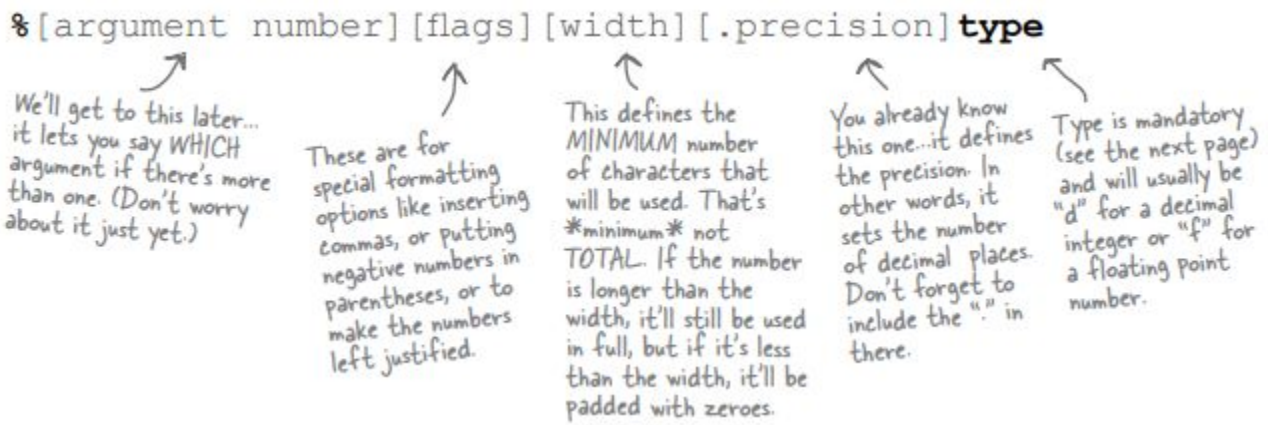


- %, d means “insert commas and format the number as a decimal integer.”
- %.2f means “format the number as a floating point with a precision of two decimal places.”

- The format String uses its own little language syntax.
- 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.
- %,.2f means “insert commas and format the number as a floating point with a precision of two decimal places.”

Number formatting. (6/n)

- The format specifier.
- 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.



What about dates? (1/n)

- 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).
- The examples below should give you a good idea of how it works:

The complete date and time **%tc**

```
String.format("%tc", new Date());
```

```
Sun Nov 28 14:52:41 MST 2004
```

Just the time **%tr**

```
String.format("%tr", new Date());
```

```
03:01:47 PM
```

What about dates? (2/n)

Day of the week, month and day **%tA %tB %td**

There isn't a single format specifier that will do exactly what we want, so we have to combine three of them for day of the week (%tA), month (%tB), and day of the month (%td).

```
Date today = new Date();
String.format("%tA, %tB %td",today,today,today)
```

The comma is not part of the formatting... it's just the character we want printed after the first inserted formatted argument

```
Sunday, November 28
```

But that means we have to pass the Date object in three times, one for each part of the format that we want. In other words, the %tA will give us just the day of the week, but then we have to do it again to get just the month and again for the day of the month.

Same as above, but *without duplicating the arguments* **%tA %tB %td**

```
Date today = new Date();
String.format("%tA, %<tB %<td",today);
```

You can think of this as kind of like calling three different getter methods on the Date object, to get three different pieces of data from it.

The angle-bracket "<" is just another flag in the specifier that tells the formatter to "use the previous argument again." So it saves you from repeating the arguments, and instead you format the same argument three different ways.

What about dates? (3/n)

- Moving backward and forward in time.
  - For a time-stamp of “now”, use Date. But for everything else, use Calendar.

- The Date class is still great for getting a “time stamp”—an object that represents the current date and time, so use it when you want to say, “give me NOW”.
- Use java.util.Calendar for your date manipulation
- More interesting, though, is that the kind of calendar you get back will be appropriate for your locale. Much of the world uses the Gregorian calendar, but if you’re in an area that doesn’t use a Gregorian calendar you can get Java libraries to handle other calendars such as Buddhist, or Islamic or Japanese.

What about Calendar Api ? (1/n)

- Getting an object that extends Calendar.
- How in the world do you get an “instance” of an abstract class? Well you don’t of course, this won’t work:

This WON’T work:

```
Calendar cal = new Calendar();
```

The compiler won’t allow this!

Instead, use the static “getInstance()” method:

```
Calendar cal = Calendar.getInstance();
```

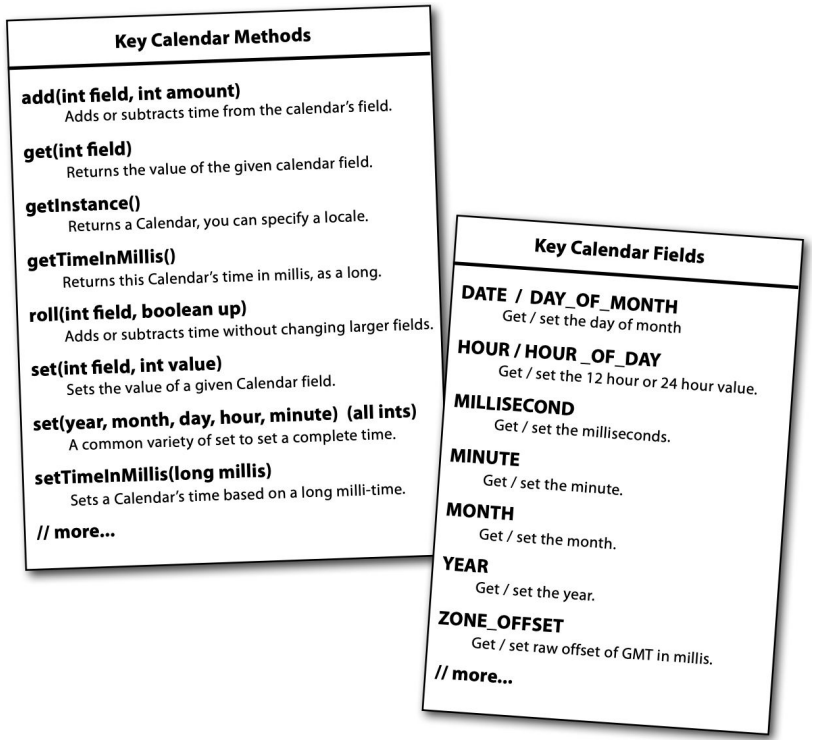
This syntax should look familiar at this point – we’re invoking a static method.

What about Calendar Api ? (2/n)

- Working 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.

What about Calendar Api ? (3/n)

- Highlights of the Calendar API.

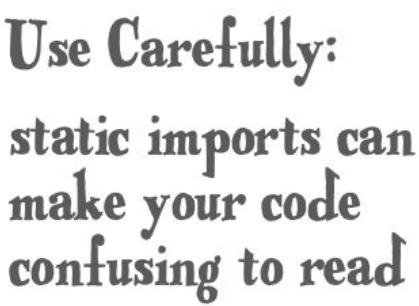


Static import (1/n)

- Even more Statics!... static imports.
- New to Java 5.0... a real mixed blessing. Some people love this idea, some people hate it. Static imports exist only to save you some typing. If you hate to type, you might just like this feature. The downside to static imports is that - if you're not careful - using them can make your code a lot harder to read.

Some old-fashioned code:

```
import java.lang.Math;

class NoStatic {

  public static void main(String [] args) {

    System.out.println("sqrt " + Math.sqrt(2.0));

    System.out.println("tan " + Math.tan(60));

  }

}
```



Static import (2/n)

Same code, with static imports:

```
import static java.lang.System.out;
import static java.lang.Math.*;
class WithStatic {
  public static void main(String [] args) {
    out.println("sqrt " + sqrt(2.0));
    out.println("tan " + tan(60));
  }
}
```

The syntax to use when declaring static imports.

Static imports in action.

Caveats & Gotchas

- If you're only going to use a static member a few times, we think you should avoid static imports, to help keep the code more readable.
- If you're going to use a static member a lot, (like doing lots of Math calculations), then it's probably OK to use the static import.
- Notice that you can use wildcards (.*), in your static import declaration.
- A big issue with static imports is that it's not too hard to create naming conflicts. For example, if you have two different classes with an "add()" method, how will you and the compiler know which one to use?

Reference

1. Head First book (page 273- 314)

Thank you!

Presented by **Jahongir Sherjonov**

(jakhongirsherjonov@gmail.com)