Statics and formatting numbers

Overview

- Static keyword
  - Static methods
  - Static instance variables
- Formatting numbers
  - printf style formatting
  - Output immediately to standard output
  - Output to a String variable

main method dissected

```
public class HelloWorld
{
   public static void main(String [] args)
   {
      System.out.println("Hello world!");
   }
}
```

	What it means
`public`	Visibility modifier, public = anybody inside or outside the class can call
`static`	**A method that promises not to use any instance variables of its class.**
`void`	Return type, void means this method has no return value.
`main`	Method name, by convention the method called "main" is what gets run when you do "java MyClassName"
`String []`	Type of the first parameter of the method, an array of Strings
`args`	Local variable name used in main() to refer to the first method parameter

Static vs. non-static methods

- Static methods
  - Forbidden from using instance vars of the class
- Non-static methods (instance methods)
  - Allowed to use instance vars
- A class can have:
  - All static methods
    - e.g. Math class
  - All non-static methods
  - A mix of both
    - e.g. Wrapper classes like Double and Integer

Math.java

Method signature	Description
`public static int round(float a)`	Round to the nearest integer
`public static long round(double a)`	
`public static int min(int a, int b)`	Find the minimum of two numbers
`public static long min(long a, long b)`	
`public static float min(float a, float b)`	
`public static double min(double a, double b)`	
`public static int abs(int a)`	Return absolute value of a number
`public static long abs(long a)`	
`public static float abs(float a)`	
`public static double abs(double a)`	
`public static double random()`	Get a random number in [0.0, 1.0)
`public static double sqrt(double a)`	Take the square root of a number

A selection of the static methods in Math.java

Math.java

```
public static final double E  = 2.718281828459045235 4;
public static final double PI = 3.14159265358979323846;
```

Constants in Math.java.

```
/**
  * Don't let anyone instantiate this class.
 */
private Math() {}
```

The constructor of the Math class.

A private constructor. Makes it impossible to actually create an object of type Math.

All calls via static method calls, e.g. Math.sqrt(), Math.min()

Using Math class

```
int x = Math.round(42.2);
int y = Math.min(56, 12);
int z = Math.abs(-343);
```

These methods don't use instance variables, so there is no specific object to call the methods on. Use class name instead.

```
Math mathObj = new Math();
```

You can't do this since the constructor is private:

```
Exception in thread "main" java.lang.Error: Unresolved
compilation problem:
        The constructor Math() is not visible
```

There is no persistent state in the Math object, so no need to create an object on the heap.

Limits of static methods

- Static methods
  - Cannot use non-static instance variables

```
Cow cow1 = new Cow();
Cow cow2 = new Cow();
System.out.println("weight = ", Cow.getWeight());
```

The weight of which cow?

```
public class Cow
{
   private String name = "";
   private double weight = 0.0;

   public static double getWeight()
   {
      return weight;
   }
}
```

Since `getWeight()` is declared static, any use of a non-static instance variable results in a compile error:

```
Cow.java:8: non-static variable weight cannot be
referenced from a static context
         return weight;
                ^
```

Limits of static methods

- Static methods
  - Cannot call non-static methods

```
public class Cow
{
   private String name = "";
   private double weight = 0.0;

   public double getWeight()
   {
      return weight;
   }

   public static boolean isBig()
   {
      return (getWeight() > 500.0);
   }
}
```

Since `isBig()` is declared static, any use of a non-static instance method results in a compile error:

```
Cow.java:13: non-static method
getWeight() cannot be referenced from
a static context
         return (getWeight() > 500.0);
                 ^
```

Wrapper classes

- Goal: Reverse doubles read from StdIn
- Problem: We don't know how many numbers

`Double` class wraps a primitive `double` data type into an object so we can put it into the `ArrayList`.

```
import java.util.ArrayList;

public class ReverseNums
{
   public static void main(String[] args)
   {
      ArrayList<Double> nums = new ArrayList<Double>();

      while (!StdIn.isEmpty())
         nums.add(StdIn.readDouble());

      for (int i = nums.size() - 1; i >= 0; i--)
         System.out.println(nums.get(i));
   }
}
```

Wrapper classes

- Wrapper classes
  - Contain both static and instance methods
  - Can create objects of a wrapper type
    - Directly via new
    - Java 5.0+, using autoboxing

Primitive type	Wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

```
int i = 288;
Integer iWrap = new Integer(i);
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(iWrap);
```

Create an object of type `Integer` to put into our `ArrayList`

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(288);
```

Java automatically creates an object of type `Integer`

Integer, selected instance methods

Constructor Summary
`Integer(int value)` Constructs a newly allocated `Integer` object that represents the specified `int` value.
`Integer(String s)` Constructs a newly allocated `Integer` object that represents the `int` value indicated by the `String` parameter.

Method Summary	
byte	`byteValue()` Returns the value of this `Integer` as a `byte`.
int	`compareTo(Integer anotherInteger)` Compares two `Integer` objects numerically.
double	`doubleValue()` Returns the value of this `Integer` as a `double`.
boolean	`equals(Object obj)` Compares this object to the specified object.

Note how we call the method, variable name dot method

```
Integer num1 = new Integer(288);
Integer num2 = new Integer("288");
if (num1.equals(num2))
    System.out.println("They are equal!");
```

Integer, selected static methods

Method Summary	
static int	lowestOneBit(int i) Returns an int value with at most a single one-bit, in the position of the lowest-order ("rightmost") one-bit in the specified int value.
static int	numberOfLeadingZeros(int i) Returns the number of zero bits preceding the highest-order ("leftmost") one-bit in the two's complement binary representation of the specified int value.
static int	numberOfTrailingZeros(int i) Returns the number of zero bits following the lowest-order ("rightmost") one-bit in the two's complement binary representation of the specified int value.
static int	parseInt(String s) Parses the string argument as a signed decimal integer.

Note how we call the method, class name name dot method

```
int N = Integer.parseInt(args[0]);
System.out.println("N = " + N);
```

Integer, selected static variables

Field Summary	
static int	MAX_VALUE A constant holding the maximum value an int can have, $2^{31}-1$.
static int	MIN_VALUE A constant holding the minimum value an int can have, -2^{31}.

```
int [] vals = {3, 2, 5, 4, 1};
int max = Integer.MIN_VALUE;
for (int i = 0; i < vals.length; i++)
{
   if (vals[i] > max)
      max = vals[i];
}
System.out.println("max = " + max);
```

Finding the maximum in an array of values.

Static variables

- Static instance variable
  - One variable shared among all instances of a particular class
  - Why?
    - Commonly used for constants

```
public static final double PI = 3.14159265358979323846;
```

    - Eliminate repeated variables that are always the same
      - Reduce memory needed to create many objects of same type
    - Track a single value related to a class type
      - e.g. Number of objects created for a particular class

Reducing memory usage

```
public class Cow
{
   private String name = "";
   private double weight = 0.0;


   private static AudioFile sound = new AudioFile("cow.wav");

   public void makeNoise()
   {
      sound.play();
   }
}
```

Normal instance variables, each `Cow` object has its own set of these variables.

All `Cow` objects share the same sound object. Only one `AudioFile` object is created on the heap, thus saving memory.

Counting created objects

```
public class Cow
{
    private String name = "";
    private double weight = 0.0;

    private static int numCows = 0;

    public Cow()
    {
        numCows++;
    }

    public Cow(Cow otherCow)
    {
        this.name   = otherCow.name;
        this.weight = otherCow.weight;
        numCows++;
    }

    public static int getNumCows()
    {
        return numCows;
    }
}
```

Normal instance variables, each `Cow` object has its own set of these variables.

No matter how many `Cow` objects we create, there is only ever one `numCows` instance variable that has a single value.

Methods that only use static variables can be declared static.

Pretty text formatting

- printf-style formatting
  - Common way to nicely format output
  - Present in many programming languages
    - Java, C++, Perl, PHP, ...
  - Use a special format language:
    - Format string with special codes
    - One or more variables get filled in
- In Java, used via:
  - System.out.printf() – output to standard output
  - String.format() – returns a formatted String

Floating-point formatting

```
double d = 0.123456789;
float  f = 0.123456789f;

// %f code is used with double or float variables
// %f defaults to rounding to 6 decimal places
System.out.printf("d is about %f\n", d);
System.out.printf("f is about %f\n", f);
```

```
d is about 0.123457
f is about 0.123457
```

\n means line feed

```
// Number of decimal places specified by .X
// Output is rounded to that number of places
System.out.printf("PI is about %.1f\n", Math.PI);
System.out.printf("PI is about %.2f\n", Math.PI);
System.out.printf("PI is about %.3f\n", Math.PI);
System.out.printf("PI is about %.4f\n", Math.PI);
```

```
PI is about 3.1
PI is about 3.14
PI is about 3.142
PI is about 3.1416
```

```
// %e code outputs in scientific notation
// .X specifies number of significant figures
final double C = 299792458.0;
System.out.printf("speed of light = %e\n", C);
System.out.printf("speed of light = %.3e\n", C);
```

```
C = 2.99792e+08
C = 3.00e+08
```

Integer formatting

```
// %d code is for integer values, int or long
// Multiple % codes can be used in a single printf()
long power = 1;
for (int i = 0; i < 8; i++)
{
   System.out.printf("%d = 2^%d\n", power, i);
   power = power * 2;
}
```

You can have multiple % codes that are filled in by a list of parameters to printf()

```
1 = 2^0
2 = 2^1
4 = 2^2
8 = 2^3
16 = 2^4
32 = 2^5
64 = 2^6
128 = 2^7
```

```
// A number after the % indicates the minimum width
// Good for making a nice looking tables of values
power = 1;
for (int i = 0; i < 8; i++)
{
   System.out.printf("%5d = 2^%d\n", power, i);
   power = power * 2;
}
```

Minimum width of this field in the output. Java will pad with whitespace to reach this width (but can exceed this width if necessary).

```
    1 = 2^0
    2 = 2^1
    4 = 2^2
    8 = 2^3
   16 = 2^4
   32 = 2^5
   64 = 2^6
  128 = 2^7
```

Flags

```
// Same table, but left justify the first field
power = 1;
for (int i = 0; i < 8; i++)
{
   System.out.printf("%-5d = 2^%d\n", power, i);
   power = power * 2;
}
```

- flag causes this field to be left justified

```
1     = 2^0
2     = 2^1
4     = 2^2
8     = 2^3
16    = 2^4
32    = 2^5
64    = 2^6
128   = 2^7
```

```
// Use commas when displaying numbers
power = 1;
for (int i = 0; i < 17; i++)
{
   System.out.printf("%,5d = 2^%d\n", power, i);
   power = power * 2;
}
```

, flag causes commas between groups of 3 digits

```
    1 = 2^0
    2 = 2^1
    4 = 2^2
    8 = 2^3
   16 = 2^4
   32 = 2^5
   64 = 2^6
  128 = 2^7
  256 = 2^8
  512 = 2^9
1,024 = 2^10
2,048 = 2^11
4,096 = 2^12
8,192 = 2^13
16,384 = 2^14
32,768 = 2^15
65,536 = 2^16
```

Text formatting

```
// Characters can be output with %c, strings using %s
String name = "Bill";
char grade = 'B';
System.out.printf("%s got a %c in the class.\n", name, grade);
```

```
Bill got a B in the class.
```

```
// This prints the same thing without using printf
System.out.println(name + " got a " + grade + " in the class.");
```

An equivalent way to print the same thing out using good old println().

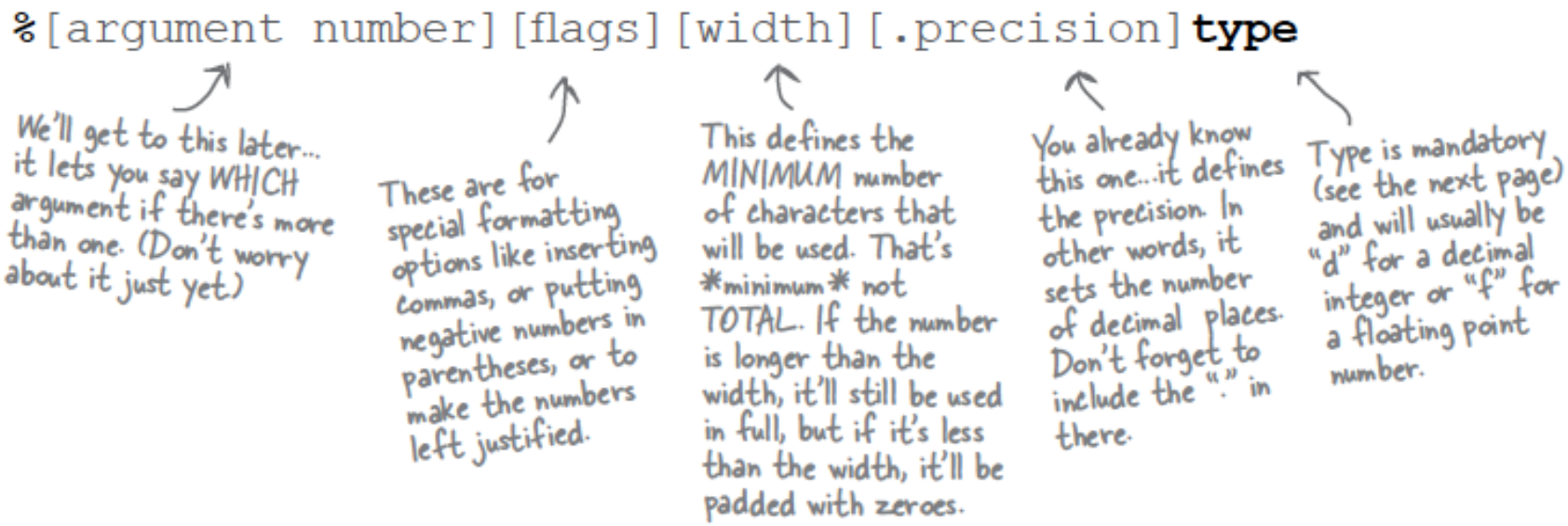
Creating formatted strings

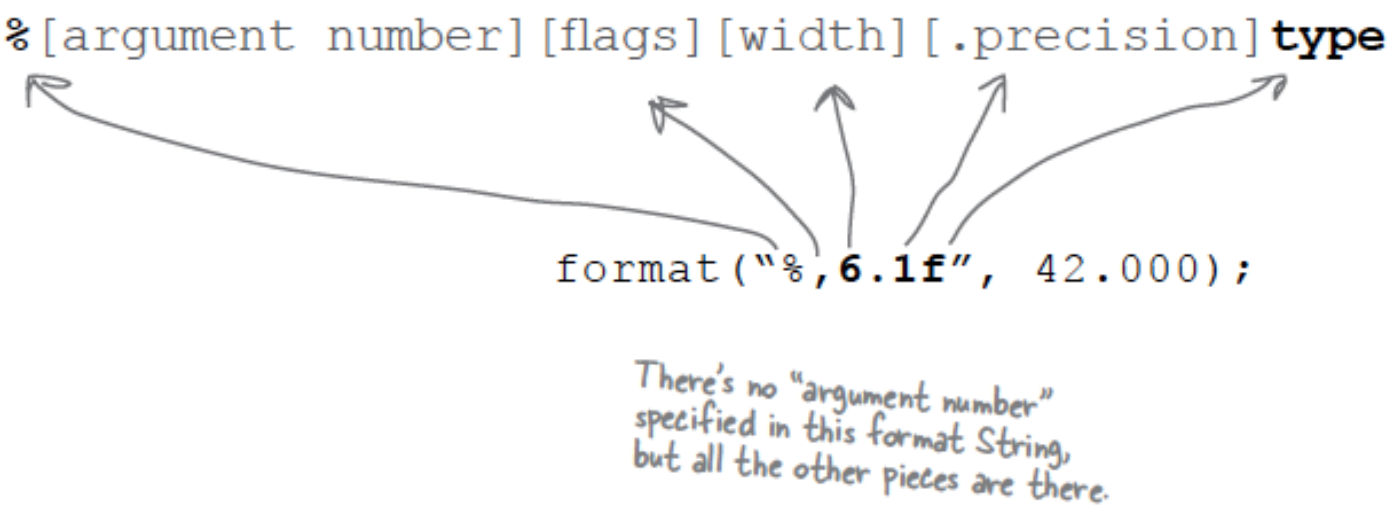
- Formatted String creation
  - You don't always want to immediately print formatted text to standard output
  - Save in a String variable for later use

```
// Formatted Strings can be created using format()
String lines = "";
for (int i = 0; i < 4; i++)
   lines += String.format("Random number %d = %.2f\n", i, Math.random());
System.out.print(lines);
```

```
Random number 0 = 0.54
Random number 1 = 0.50
Random number 2 = 0.39
Random number 3 = 0.64
```

The format specifier





printf gone wild

- Format string specifies:
  - Number of variables to fill in
  - Type of those variables
- Make sure format string and agrees with arguments afterwards
  - Runtime error otherwise
  - Compiler / Eclipse won't protect you

```
// Runtime error %f expects a floating-point argument
System.out.printf("crash %f\n", 1);

// Runtime error, %d expects an integer argument
System.out.printf("crash %d\n", 1.23);

// Runtime error, not enough arguments
System.out.printf("crash %d %d\n", 2);
```

printf puzzler

Code	Letter
`System.out.printf("%f", 4242.00);`	
`System.out.printf("%d", 4242);`	
`System.out.printf("%.2f", 4242.0);`	
`System.out.printf("%.3e", (double) 4242);`	
`System.out.printf("%,d", 4242);`	

Letter	Output
A	`4242`
B	`4242.00`
C	`4.242e+03`
D	`4,242`
E	`4242.000000`

Code	#
`System.out.printf("%d%d", 42, 42);`	
`System.out.printf("%d+%d", 42, 42);`	
`System.out.printf("%d %d", 42);`	
`System.out.printf("%8d", 42);`	
`System.out.printf("%-8d", 42);`	
`System.out.printf("%d", 42.0);`	

#	Output
1	`42+42`
2	`4242`
3	`  42`
4	`42  `
5	`runtime error`

printf puzzler

Code	Letter
`System.out.printf("%f", 4242.00);`	E
`System.out.printf("%d", 4242);`	A
`System.out.printf("%.2f", 4242.0);`	B
`System.out.printf("%.3e", (double) 4242);`	C
`System.out.printf("%,d", 4242);`	D

Letter	Output
A	`4242`
B	`4242.00`
C	`4.242e+03`
D	`4,242`
E	`4242.000000`

Code	#
`System.out.printf("%d%d", 42, 42);`	
`System.out.printf("%d+%d", 42, 42);`	
`System.out.printf("%d %d", 42);`	
`System.out.printf("%8d", 42);`	
`System.out.printf("%-8d", 42);`	
`System.out.printf("%d", 42.0);`	

#	Output
1	`42+42`
2	`4242`
3	`  42`
4	`42`
5	`runtime error`

printf puzzler

Code	Letter
`System.out.printf("%f", 4242.00);`	E
`System.out.printf("%d", 4242);`	A
`System.out.printf("%.2f", 4242.0);`	B
`System.out.printf("%.3e", (double) 4242);`	C
`System.out.printf("%,d", 4242);`	D

Letter	Output
A	`4242`
B	`4242.00`
C	`4.242e+03`
D	`4,242`
E	`4242.000000`

Code	#
`System.out.printf("%d%d", 42, 42);`	2
`System.out.printf("%d+%d", 42, 42);`	1
`System.out.printf("%d %d", 42);`	5
`System.out.printf("%8d", 42);`	3
`System.out.printf("%-8d", 42);`	4
`System.out.printf("%d", 42.0);`	5

#	Output
1	`42+42`
2	`4242`
3	`  42`
4	`42  `
5	`runtime error`

Summary

- Static methods
  - A method that doesn't rely on state of an object
    - Uses only its input parameters to produce its output
    - e.g. Math methods sqrt(), abs(), pow()
  - Can mix static and instance methods
    - e.g. Wrapper classes: Integer, Double, ...
- Static instance variables
  - Shared variable between all objects of a class
- Nicely formatting output
  - System.out.printf(String format, ...)
  - String.format(String format, ...)