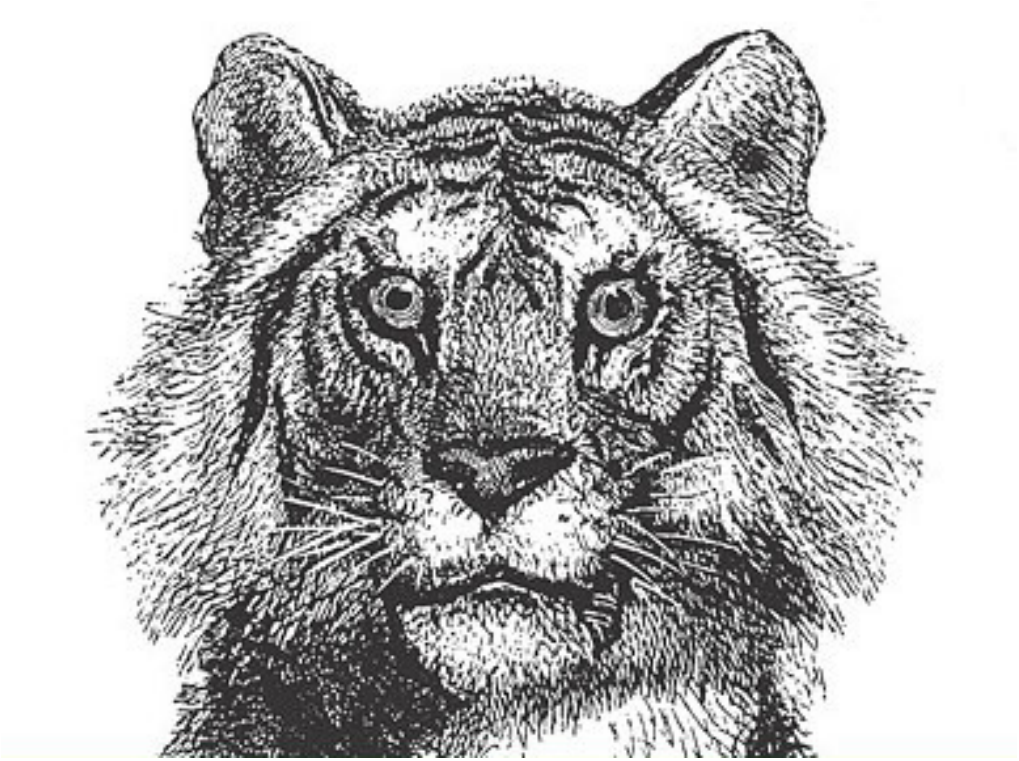
MORE BASIC JAVA

Today

- Today we will continue to look at some of the basic aspects of Java, including:
  - Data types
  - Objects (quickly)
  - Control structures
  - Exception handling
- We will also constrast some of these aspects with C++.



- Again, lots of this material is drawn from *Java in a Nutshell*, David Flanagan, O'Reilly …

`main`

- Remember `HelloWorld`:

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

- The argument `(String[] args)` allows access to the parameters of the program when it is run from the command-line.
- (This is the parameter that is conventionally called `argv` in C++)
- If we want the number of arguments, we can get that with:

```
args.length()
```

Comments

- You should have noticed comments in the programs we looked at last time.
- These began with:

```
// Here is a comment
```

- As in C++, we can also have comments between `/*` and `*/`:

```
/* Here is a multi-line comment
   which continues
   until we get to this end comment symbol */
```

- Comments using `/*` and `*/` do not nest.
- So, a good suggestion is to mainly use `//` for short comments so that you can comment out large sections with `/*` and `*/` do not nest when debugging.

- Java also supports a special "doc comment":

```
/** Here is doc comment
    which continues
    until we get to this end comment symbol */
```

- The `javadoc` tool constructs simple online documentation from these comments.

Constants

- The keyword `final` prevents a variable having its value changed.
  - It becomes a constant
- So values have to be given to such variables when they are defined:

```
public static final double PI = 3.1415927;
```

- By convention constants are given names in CAPITALS.
- The compiler uses constant values, where possible, to compute values at compile-time.

Primitive data types

- In Java, all variable have guranteed default values
  - The compiler might warn you about relying on them.
- Basic types of types:
  - Boolean
  - Character
  - Integral (integer) value
  - Floating-point

`boolean`

- Contains: `true` or `false`
- Default: `false`
- Size: 1 bit.
- Note that these are NOT integers — have to convert between integers and `booleans`

`char`

- Contains: Unicode character
- Default: `\u0000`
- Size: 16 bits.
- Range: `\u0000` to `\uFFFF`

- Can write character constants in programs between single quotes:

```
char c = 'A';
```

- Standard C/C++ escape characters work:

```
char newline = '\n';
```

- Since Java uses Unicode for characters, Unicode escape sequences work as well:

```
char aleph = '\u0500';
```

`byte`

- Contains: Signed integer
- Default: 0
- Size: 8 bits.
- Range: -128 to 127

`short`

- Contains: Signed integer
- Default: 0
- Size: 16 bits.
- Range: -32768 to 32767

`int`

- Contains: Signed integer
- Default: 0
- Size: 32 bits.

`long`

- Contains: Signed integer
- Default: 0
- Size: 64 bits.

- No unsigned integral values
- Not legal to write:

```
long int
```

```
short int
```

- Division or modulo by zero throws an exception

`float`

- Contains: Floating point (IEEE 754)
- Default: 0.0
- Size: 32 bits.

`double`

- Contains: Floating point (IEEE 754)
- Default: 0.0
- Size: 64 bits.

- Floating point literals are written as in C/C++:

```
double a = 2.0;
```

- These can also be indicated by using `d`, `D`:

```
double b = 2d;
```

- At this point `float` is pretty much deprecated and you'll find some weird behavior if you use `float`.

`Integer`

- Each primitive datatype has an associated class.
  - The class for `int` is `Integer`
- This class provides us with some useful functionality that works with the datatype.
- For example, we'll need to convert integers to strings that represent the integer value.
  - That is we'll need the string `"5"` from the integer `5`.

```
a = 5;
Integer aInt = 5;
g.drawString("a = " + aInt.toString(), 10, 20);
```

Reference types

- Java does not have pointers.
  - But it (naturall) still needs references.
- All primitive types are handled “by value”.
  - When you declare a variable, that variable holds the relevant value.
  - A copy of that value is passed to methods.
- Non-primitive types (objects, arrays) are handled “by reference”
  - When you declare a variable it holds a reference (address).
  - A copy of this reference is passed to methods.

- It is not uncommon to have two (reference) variables refer to the same object:

```
JButton p, q;
p = new JButton();
q = p;
```

- So, if we change `p`, we also change `q`:

```
p.setLabel("Ok");
String s = q.getLabel();
```

Objects (quickly)

- Declaring a variable that is an object doesn't create the object:

```
JButton b;
```

- The `new` keyword is required to create the object:

```
b = new JButton();
```

- Here the variable `b` is a reference to the object.
- Can also use the `newInstance()` method or by "deserializing" an object.

- Objects of type `String` can be created as follows:

```
String s = "metamorphosis";
```

- We access objects using the dot notation.

```
Point p = new Point();
p.x = 2.0;
p.y = 3.0;
```

- Very similar notation is used for methods:

```
Rectangle r = new Rectangle();
:
:
double theArea = r.area();
```

- In Java we don't have to delete objects.
- Instead, Java uses *garbage collection* to detect objects that are not longer being used.
  - Objects with no more references to them.
- Such objects are automatically deleted.
- So we don't have to worry about this.

Strings

- Strings are instances of the `java.lang.String` class.
- Created and assigned just like primitive variables:

```
String s;
s = "hello world!";
```

- String values can't be changed
  - Need to create a `StringBuffer`, play with that, and then make a new `String` from it.
- Lots of useful methods that you can call on `String`s:
  - `length()`
  - `equals()`
  - `substring()`
  - `...`

Arrays

- Arrays are a lot like objects:
  - Manipulated by reference.
  - Dynamically created by `new`
  - Garbage collected.
- Some details follow.

- One way to create an array is to reserve space for it and fill in the values later:

```
JButton buttons[] = new JButton[10];
```

- This can be broken into two parts:

```
JButton buttons[];
buttons = new JButton[10];
```

- Allowing us to change our mind:

```
buttons = new JButton[20];
```

- It may helpto think of this as the difference between defining a C++ array, and defining a pointer to a type that is then assigned the address of an array.

- The other way to create an array is to use an *initializer*:

```
int lookupTable[] = {1, 2, 4, 8, 16, 32, 64};
```

- We access array elements using the same notation as in C/C++

```
int lookupTable[] = {1, 2, 4, 8, 16, 32, 64};

int a = lookupTable[3];
```

- As always, the first element of the array has number 0;

Operators

- The set of Java operators looks a lot like the set of C/C++ operators.
- `instanceof`

  allows you to check if an object is an instance of a class (or implements a given interface).
- `+`

  can be used to concatenate `Strings`
- Note that Java doesn't do operator overloading.

Control structures

- `if`, `if/else`, `while` and `do/while` statements are pretty much as you know them from C/C++.
- You can't get away with some of the tricks you may have learnt in C/C++ tho'.
  - The conditional expression (the bit inside the parentheses) has to be a `boolean`.
  - So, this:

```
int i = 0;
while(i) {
  i++;
}
```

    would not be allowed.

- The `switch` statement can use:
  - `byte`
  - `char`
  - `short`
  - `int`
  - `long`

  as the `case` labels.

- `for` loops allow multiple variables to be handled:

```
int i, j;
for(i=0, j=0; i < j; i++, j--){
   System.out.println(i+j);
}
```

- These variable can also be declared in the loop

```
for(int i=0, j=0; i < j; i++, j--){
   System.out.println(i+j);
}
```

in which case their scope is the loop itself.

for/in

- Java includes a new form of `for` loop.
- Given

```
int lookupTable[] = {1, 2, 4, 8, 16, 32, 64};
```

we could define:

```
for(int n : lookupTable){
  System.out.println(n)
}
```

which would print out each element of the array in turn.

- In other words the `for` instantiates `n` with each element of the array in turn.

- It is customary to read the `:` as “in”
  - Hence the name of the construct
- Thus

```
for(int n : lookupTable){
  System.out.println(n)
}
```

  is read “for int n in loookupTable”.
- Note that the type of the loop variable (`n`) has to match the type of the array elements.

Methods

- Methods look much like they do in C/C++, for example:

```
double distanceFromOrigin(double x, double y){
  return Math.sqrt(x*x + y*y);
}
```

- There are additional keywords (modifiers) that can be placed in front of the return type
  - We will get to these later.

- One nice thing that Java makes easy is writing functions that take variable numbers of arguments:

```
int max(int first, int... rest){
  int max = first;
  for (int i: rest){
    if (i > max){
       max = i
    }
    return max;
}
```

- This says that `max` has one argument that is an integer and zero or more other integer arguments.

- All these are reasonable ways to call `max`:

```
max(1);
max(1, 2);
max(1, 2, 3);
```

- Clearly those other arguments, if they exist, are handled like an array.

Summary

- We have covered some of the basic syntax of Java, and some of the building blocks from which we can construct classes.
  - This looks a lot like C/C++, with some minor differences.
- Perhaps the biggest is the absence of pointers
  - Though references give us much the same behavior without the same possibility for error.
- The next step is to look at creating classes and objects.
  - We will start that next week.