CSE 1325: Object-Oriented Programming

Lecture 04

Class Members

Mr. George F. Rice

george.rice@uta.edu

Office Hours:
Prof Rice 12:30 Tuesday and Thursday in ERB 336
For TAs see this web page

I named my new band 1023MB. We haven't quite had a gig yet.

Today's Topics

- OOP Definition & Vocabulary
- Class Members
  - Constructors (and Destructors)
  - Methods and Fields
  - toString() and main()
  - == vs equals()
  - static and final
- More about Ant and build.xml
- Intellectual Property

What's *this*, anyway?

Image by dangkhoa1848 of Pixabay – No attribution is required
https://pixabay.com/photos/choir-choir-master-church-conductor-5781096/

Definitions of OOP

- One perspective is by *capabilities* or *language features*

  Object-Oriented Programming =

  Encapsulation (controlling access to private data)

  + **Inheritance** (reusing and extending encapsulated data)

  + **Polymorphism** (dynamically selecting encapsulated behavior)

Coming Later!

- Another perspective is by *structure*
  - A structured program is organized around actions and logic
    - "The algorithm's the thing", with apologies to The Bard
  - An object-oriented program is organized around **data** encapsulated in **classes** that also contain related **methods**
    - "The data's the thing"

Pie photo by Evan-Amos [Public domain]
https://commons.wikimedia.org/wiki/File:Cherry-Pie-Slice.jpg

Object-Oriented Terminology

Class
A template, like a cookie cutter, used to create "objects"

Object
Code and encapsulated data created from a class, sometimes called an "**Instance**"*

Variable
A symbolic name that references an object

Encapsulation
Bundling data and related code ("**Methods**") into a restricted container (a "class")

Memory
A system to store objects for rapid access

Operator
A symbol that modifies an object, or generates a new object from an existing object

* An object is an *instance* of a class like an operating system task is an *instance* of an executable file.

Cookie cutter image is CC0 Public Domain
https://www.rawpixel.com/image/5924435/photo-image-christmas-public-domain-shape

Object-Oriented Terminology

The Boring List Version

- **Encapsulation** – bundling data and code into a restricted container
- **Class** – a template encapsulating data and code that manipulates it
- **Method** – a function that manipulates data in a class
- **Instance** – an encapsulated bundle of data and code
- **Object** – an instance of a class containing a set of encapsulated data and associated methods
- **Variable** – a block of memory associated with a symbolic name that contains an object or a primitive data value
  - Variables within a class are called “fields” (or “attributes” or “class variables”)
- **Operator** – a short string representing a mathematical, logical, or machine control action

These are the actual definitions you'll need to know for the first exam.

Classes

- A class **encapsulates** data and associated code
- A class *directly represents* a concept in a program
  - If you can think of “it” as a separate entity, it is plausible that it could be a class or an object of a class
  - Examples: vector, matrix, input stream, string, FFT, valve controller, robot arm, device driver, picture on screen, dialog box, graph, window, temperature reading, clock
- **A class is a user-defined *type* that specifies how objects of its type can be *created, used,* and *destroyed***
- In Java (as in most modern languages), a class is the key building block for large programs, and very useful for small ones, too!
- The concept was originally introduced in Simula67 – in 1967!
- The class’ **public** methods and fields are its **interface**

A Class is a *Custom Type*

- Some types are *primitive* – they represent data that is directly manipulated in the hardware
  - int and its variants, e.g., short and long – char
  - double and its cousin float – boolean
- Some types are *classes* – they represent private data structures and the code that manipulates them
  - String from the standard library (a special class in Java)
  - ArrayList, HashMap, et. al. from the standard library
- The classes (and enums) you write are just as much a type as any of the above!

Creating a Variable for an Object

- In C you wrote this
  - `int i = 5;`
  - `int` is the type, `i` is the variable, and `5` is the value
- In Java you will now write this
  - `Foo foo = new Foo();`
  - `Foo` is the (class) type, `foo` is the variable, and `new Foo()` is the value
  - The `new` operator invokes the Foo constructor with the matching parameter set
    - The constructor *constructs* the value – a new Foo object

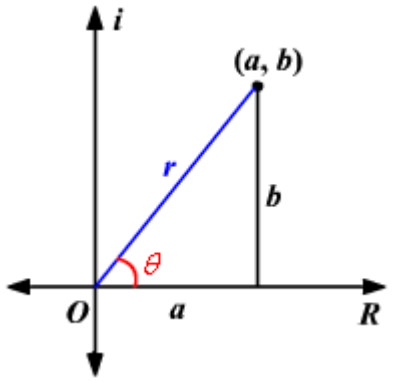
Creating a Variable: Special Cases

- String has the usual way and a special way
  - `String s = new String(char[]{'H','i'});`
  - `String s = "Hi";`
- Enum has only a special way
  - ```
    enum Dim{HEIGHT, WIDTH, DEPTH};
    Dim d = Dim.WIDTH;
    ```
    - This invokes the Dim constructor (the default in this case)
      - Note that enum constructors are *private* – invoked as shown here *without* the new keyword
    - Only one object for each element is *ever* constructed and is then shared among variables that reference it*

* The technical name for this is "Singleton"
    ```

A “Complex” Example

- Complex numbers consist of a pair of doubles
  - Real and imaginary parts, written e.g., 3.0+4.0i
  - If the imaginary part is zero, just becomes a double (e.g., 3.0+0.0i is just 3)
- Complex numbers may be represented in Cartesian or polar coordinate system
  - Cartesian is simply a + b i
  - Polar uses the magnitude r ($r^2 = a^2 + b^2$) and angle θ ($a = r \cos \theta$ and $b = r \sin \theta$) to create the form $r(\cos \theta + i \sin \theta)$
  - So $r = \sqrt{a^2 + b^2}$ and $\theta = \tan^{-1}(b/a)$



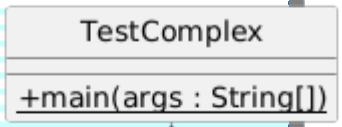
A Complex Class

- Complex numbers are useful
  - The solution (zero-intercept) of many polynomials are in the complex plane, e.g., the solution to $y=x^2+1$ is i
  - Electrical engineers represent AC as complex numbers (real is voltage, imaginary is frequency)
  - Fluid dynamics and turbulence are modeled with complex numbers, or so I am told
- Let's create a Complex Java class
  - Store a, b, and whether I/O is in Cartesian or polar
  - Provide both constructors and useful methods
  - Design decisions focus on teaching you OOP!

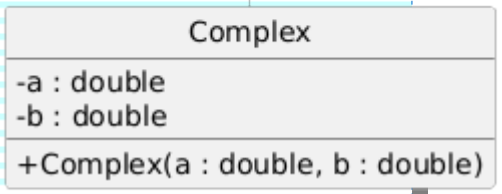
Starting the Class
(Directory complex01)

- Fields a and b are doubles in *class scope*
  - We make them private so they are *encapsulated*
- Class TestComplex gives it a trial run

```
public class Complex {
    // Constructor invoked by e.g., Complex c = new Complex(3.0, 4.0);
    public Complex(double a, double b) {
        this.a = a;   // a and b are visible everywhere in class Complex
        this.b = b;
    }
    private double a; // (real)      These fields can't be accessed
    private double b; // (imaginary)   outside of class Complex
}
```




```
public class TestComplex {
    public static void main(String[] args) {
        Complex c = new Complex(3.0, 4.0);
        System.out.println("Created Complex number " + c);
    }
```




```
ricegf@antares:~/dev/202108/04/code_from_slides/complex01$ javac TestComplex.java
ricegf@antares:~/dev/202108/04/code_from_slides/complex01$ java TestComplex
Created Complex number Complex@4d7e1886
ricegf@antares:~/dev/202108/04/code_from_slides/complex01$
```

Class Scope is Different from Local Scope

- Local scope is forward only
  - You must *declare* before you can *reference*
- Class scope is bidirectional!

```
public class Complex {
    // Constructor invoked by e.g., Complex c = new Complex(3.0, 4.0);
    public Complex(double a, double b) {
        this.a = a;   // a and b are visible everywhere in class Complex
        this.b = b;
    }
    private double a; // (real)      These fields can't be accessed
    private double b; // (imaginary)   outside of class Complex
}
```

Backward reference (permitted in class scope only)

```
public class TestComplex {
    public static void main(String[] args) {
        Complex c = new Complex(3.0, 4.0);
        System.out.println("Created Complex number " + c);
    }
```

Forward reference (local or class scope)

```
ricegf@antares:~/dev/202108/04/code_from_slides/complex01$ javac TestComplex.java
ricegf@antares:~/dev/202108/04/code_from_slides/complex01$ java TestComplex
Created Complex number Complex@4d7e1886
ricegf@antares:~/dev/202108/04/code_from_slides/complex01$
```

Printing an Object in Java

- If you do not specify otherwise, printing an object will get you name@hashcode*
  - Complex@4d7e1886 is the default in our example
  - The name Complex specifies the block of code (the methods) for this class
  - The hex number 4d7e1886 specifies the hashcode for this specific object (it's actual field values)
  - This string should uniquely specify the object
- Let's change this to the a+bi form users expect

* Specifically, `getClass().getName() + '@' + Integer.toHexString(hashCode())`

toString

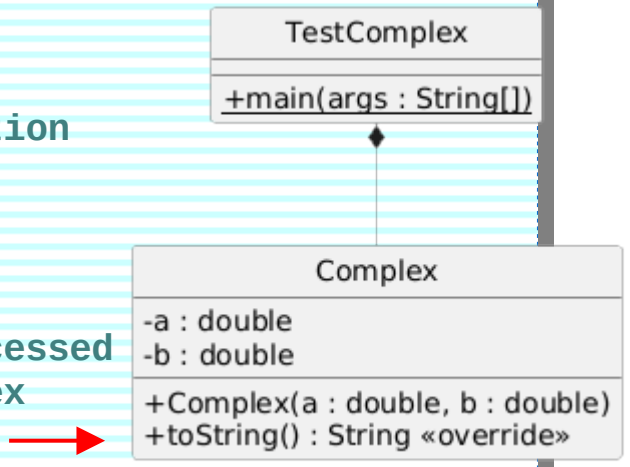
- Every object has a toString method in Java
  - Hence every object can be converted into a String!
  - Guess what it returns by default*
- We often *override* the existing toString method
  - For now, consider “override” to mean “replace”
  - `@Override` tells the compiler to check our syntax
  - We’ll provide a more formal definition in Lecture 07
- We can force a “String context”
  - `String s = "" + c;` `// force c to a String`
  - `String s = c.toString();` `// or call it directly`

* Exactly! The class name, ‘@’, and the hashcode!

Specifying a String Representation
(Directory complex02)

- We can replace (“override”) the default string representation with one of our choosing

```
public class Complex {
    // Constructor invoked by e.g., Complex c = new Complex(3.0, 4.0);
    public Complex(double a, double b) {
        this.a = a;
        this.b = b;
    }
    // toString method replaces the default string representation
    @Override
    public String toString() {
        return a + "+" + b + "i";
    }
    private double a; // (real)      These fields can't be accessed
    private double b; // (imaginary)   outside of class Complex
}
```



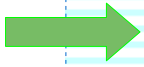
The *annotation* called @Override tells the compiler we are replacing a default method. The compiler will then throw an error if we misspell it or use the wrong parameters!

```
ricegf@antares:~/dev/202208/04/code_from_slides/complex02$ javac TestComplex.java
ricegf@antares:~/dev/202208/04/code_from_slides/complex02$ java TestComplex
Created Complex number 3.0+4.0i
ricegf@antares:~/dev/202208/04/code_from_slides/complex02$
```

Fields vs Parameters vs Locals

- Complex.Complex receives 2 doubles a & b - variables that are **parameters**.
- Complex also defined 2 doubles a & b - variables that are **fields**.
- Complex.main also defines 2 doubles a & b - variables **local** to main.





```
public class Complex {
    public Complex(double a, double b) {
        this.a = a;
        this.b = b;
    }
    public Complex() {
        this(0.0, 0.0);
    }
    @Override
    public String toString() {
        return "" + a + "+" + b + "i";
    }
    private double a; private double b;

    public static void main(String[] args) {
        double a = 3.0; double b = 4.0;
        Complex c = new Complex(a, b);
        System.out.println("" + c);
    }
}
```

Constructors

- A constructor is a special kind of member that initializes an instance of its class – a new object
  - The constructor must initialize (construct) fields, allocate memory, update static variables, etc.
  - **The keyword *`this`* references the current object**, and is very useful for specifying fields
  - So ***`this.a`*** means “the **`a`** field in the current object”, while simply ***`a`*** refers to the parameter of the same name
    - **`this.a = a;`** is a very common Java constructor idiom
- Constructors always share the name of the class
  - Constructors NEVER have a return type (not even void)
  - Constructors are usually the first class members

Constructors

- **A constructor is NOT a method** – it is a special member
- **A constructor has NO return type*** – it constructs the object
- A constructor can have any number of parameters
  - The default constructor has zero parameters
  - If no constructors are specified, the compiler defines a “free” default that just initializes the fields using *their* default constructors†
  - **If <u>any</u> constructors are specified, the compiler will NOT provide a default constructor**
- A class may have any number of constructors
  - Each must have a unique *parametric signature* – that is, the number and types of parameters must be unique to every constructor

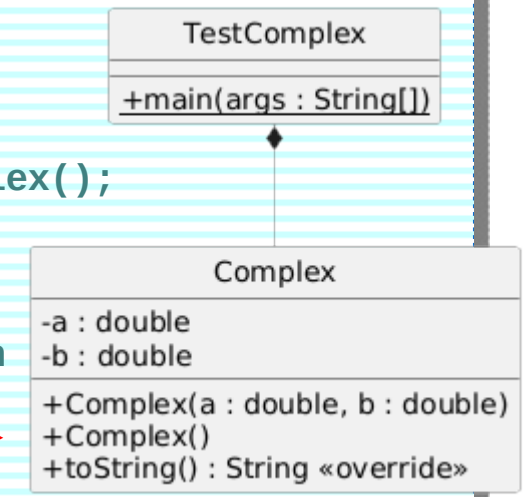
* Not even void!

† int, double, char, and bool default to 0, 0.0, \0, and false respectively

Adding a Default Constructor
(Directory complex03)

- We can explicitly define the default constructor in addition of other constructors

```
public class Complex {
    // Constructor invoked by e.g., Complex c = new Complex(3.0, 4.0);
    public Complex(double a, double b) {
        this.a = a;
        this.b = b;
    }
    // Default constructor invoked by e.g., Complex c = new Complex();
    public Complex() {
        this(0,0);    // Chain to the first constructor
    }
    // toString method replaces the default string representation
    @Override
    public String toString() {
        return a + "+" + b + "i";
    }
    private double a; // (real)      These fields can't be accessed
    private double b; // (imaginary)   outside of class Complex
}
```



If we provide NO constructors, we get a default “for free”. But if we define ANY other constructor, we do NOT get a default – we must explicitly write one if we want it!

Adding a Default Constructor

- Complex() chains to Complex(double, double)

```
public class Complex {
    // Constructor invoked by e.g., Complex c = new Complex(3.0, 4.0);
    public Complex(double a, double b) {
        this.a = a;
        this.b = b;
    }
    // Default constructor invoked by e.g., Complex c = new Complex();
    public Complex() {
        this(0,0);    // Chain to the first constructor
    } // ...
```

```
public class TestComplex {
    public static void main(String[] args) {
        Complex c1 = new Complex(3.0, 4.0);
        Complex c2 = new Complex();
        System.out.println("Created Complex numbers " + c1 + " and " + c2);
    }
}
```

```
ricegf@antares:~/dev/202208/04/code_from_slides/complex03$ javac TestComplex.java
ricegf@antares:~/dev/202208/04/code_from_slides/complex03$ java TestComplex
Created Complex numbers 3.0+4.0i and 0.0+0.0i
ricegf@antares:~/dev/202208/04/code_from_slides/complex03$
```

Constructor Chaining

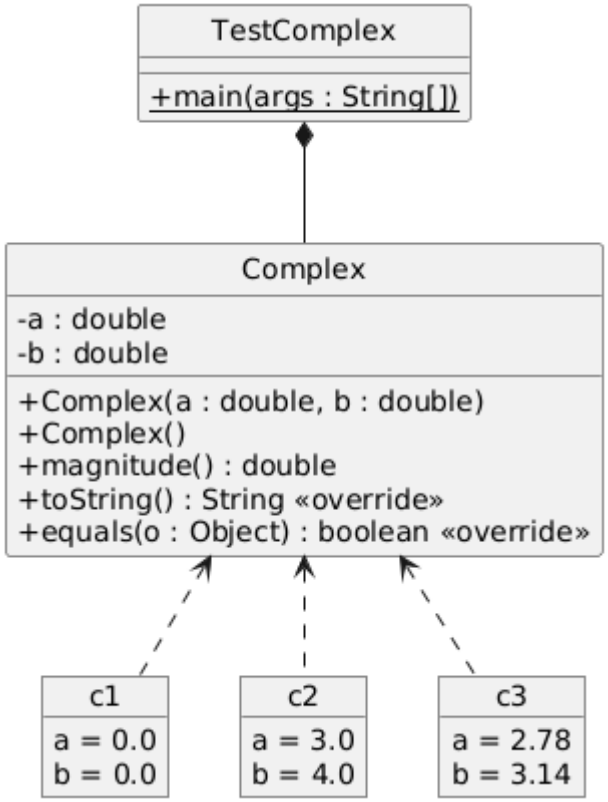
- The DRY Principle: Don't Repeat Yourself
  - Try to minimize duplication across constructors
- One way to accomplish this is via *chaining*
  - The constructor with the most parameters contains the data validation and initializations
  - Constructors with fewer parameters *chain* to it, providing default values as needed


```
public class Complex {
    // Constructor invoked by e.g., Complex c = new Complex(3.0, 4.0);
    public Complex(double a, double b) {
        this.a = a;
        this.b = b;
    }
    // Default constructor invoked by e.g., Complex c = new Complex();
    public Complex() {
        this(0,0);    // Chain to the first constructor
    } // …
```

Class vs Object

The class contains the method code and the template for the data

The objects simply fill in the data template from the class



Destructors

- The destructor tears down or deallocates the object, freeing any resources
  - In a non-memory managed language such as C++, this is a critical issue
  - Poorly written C++ destructors may result in “memory leaks”
- Java memory is *actively* managed, and freed when it can no longer be accessed
  - This is called “garbage collection”, about which we’ll have *much* more to say later
  - The result: **Java does not need or support[1] destructors**

[1]Java had a special finalize() method, but it is deprecated (its use is discouraged). For software more complex than we will create in this class, consider Cleaner and PhantomReference instead.

Another “special” method: main!

- A class is *executable* if it has a main method 
- The signature must *exactly*[1] match
  `public static void main(String[ ] args) { … }`
  - Public means “visible from outside of our class”
  - More on *static* later today
  - Java’s main is **`void`** (it returns nothing)
    - Return an int using the **`System.exit(-1);`** method
  - The single parameter **`String[] args`** is *required*
    - *Mostly* equivalent to C’s **`int main(int argc, char* argv[ ])`**
    - Java’s **`args.length`** gives us C’s argc-1, the number of arguments but *excluding* the executable name with is always the class name
    - Java’s **`args[0]`** gives us C’s argv[1], the first argument
      - C’s argv[0] is Java’s class name, also ***`object`*`.getClass().getName()`**

[1] You can add “throws” clauses, though – more on those later.

Main's Parameter

- Here's a simple Java program to print the number and value of its arguments

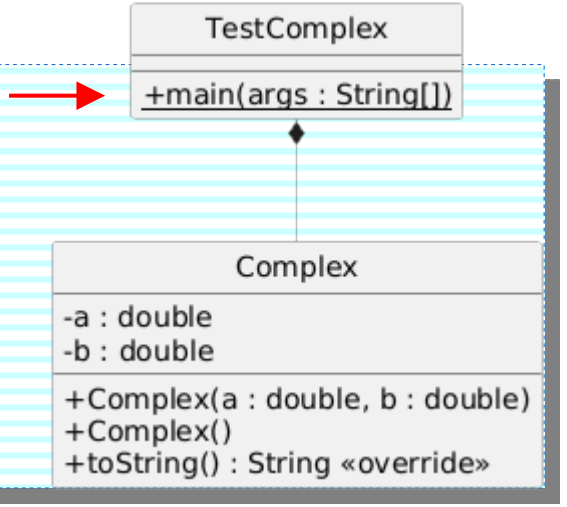
```
public class Args {
  public static void main(String[] args) {
      System.out.println("This program has " + args.length + " arguments");
      for(String s : args)
        System.out.println("  " + s);
  }
}
```

```
ricegf@antares:~/dev/202108/04/code_from_slides$ javac Args.java
ricegf@antares:~/dev/202108/04/code_from_slides$ java Args
This program has 0 arguments
ricegf@antares:~/dev/202108/04/code_from_slides$ java Args arg1 arg2 arg3
This program has 3 arguments
  arg1
  arg2
  arg3
ricegf@antares:~/dev/202108/04/code_from_slides$
```

Comparing Objects

- Let's compare two complex numbers that have the same values for a and b

```
public class TestComplex {
    public static void main(String[] args) {
        Complex c1 = new Complex(3.0, 4.0);
        Complex c2 = new Complex(3.0, 4.0);
        System.out.println("" + c1 + " "
            + ((c1 == c2) ? "==" : "!=")
            + " " + c2);
    }
}
```




```
ricegf@antares:~/dev/202108/04/code_from_slides/complex04$ javac TestComplex.java
ricegf@antares:~/dev/202108/04/code_from_slides/complex04$ java TestComplex
3.0+4.0i != 3.0+4.0i
ricegf@antares:~/dev/202108/04/code_from_slides/complex04$
```

What??? I'm going back to C!

== vs equals

- In Java, the *value* of a primitive is the value of the primitive
  - For ints i and j, `i == j` compares the values
- In Java, the *value* of an object is its address
  - For objects o1 and o2, `o1 == o2` compares their *address* rather than the *value* of the two objects (like `&o1 == &o2` in C)
  - That is, do o1 and o2 reference the exact same *object*?
- To compare contents of 2 objects, use the `equals` *method*[1]
  - By default, the `equals` method compares the addresses
  - That is, by default, `==` and `equals` function identically for objects
  - **But we can *replace* ("override") the `equals` method!**

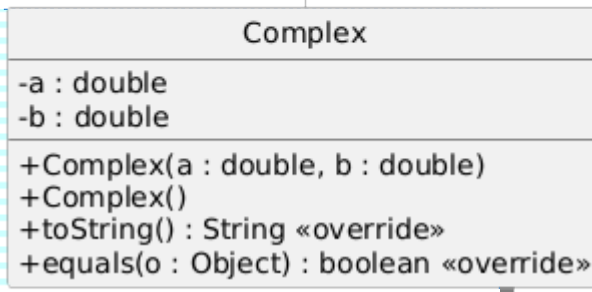
[1]Primitives in Java have no methods, thus they have no `equals` method.
They may also be on the stack or in a CPU register, and thus have no *address*, either!

Comparing Object Values
(Directory complex04)

- Let's replace the equals method for class Complex
  - The rest of the Complex class is unchanged

```
    @Override
    public boolean equals(Object o) {              // We compare this to ANY object
        if(o == this) return true;                 // An object is equal to itself
        if(o == null || getClass() != o.getClass()) // A null object isn't equal
            return false;                          // A different type isn't equal
        Complex c = (Complex)o;                    // Cast to Complex as in C
        return (a == c.a) && (b == c.b);           // Compare the values as needed
    }
```

```
public class TestComplex {
    public static void main(String[] args) {
        Complex c1 = new Complex(3.0, 4.0);
        Complex c2 = new Complex(3.0, 4.0);
        System.out.println("" + c1 + " "
            + ((c1 == c2) ? "==" : "!=") + " " + c2);
        System.out.println("" + c1 + " "
            + ((c1.equals(c2)) ? " equals " : " does not equal ") + " " + c2);
    }
```




```
ricegf@antares:~/dev/202108/04/code_from_slides/complex04$ javac TestComplex.java
ricegf@antares:~/dev/202108/04/code_from_slides/complex04$ java TestComplex
3.0+4.0i != 3.0+4.0i
3.0+4.0i  equals  3.0+4.0i
ricegf@antares:~/dev/202108/04/code_from_slides/complex04$
```

OK, I'll stick with Java a while longer!

Some thoughts on method equals

- The first line deals with `if (c1.equals(c1))`
  - Yes, a Complex instance is *always* equal to itself!
- The second line deals with `if (c1.equals("Hi!"))`
  - No, a different type is *never* equal to a Complex!
- The third line lets us access fields in the comparand
  - An object can see private fields in other objects of the same class
- The last line compares the relevant fields
- NOTE: This implementation works but is incomplete
  - In particular, it will fail if used with any class that relies on reliable hashing, e.g., HashMap
  - More on this (much) later in the course – for now, don't worry about it – close enough for Exam #1!

Four Easy Steps to Equality

- The FOUR STEPS of writing equals
  - Is it me? True!
  - Is it null or not my type? False!
  - Cast it to my type
  - Return the comparison of the key fields
- You need to remember these 4 steps for the exam AND for writing equals methods

Comparing With Other Types

- Think of Object as meaning “any non-primitive type” for now
- We can compare equality of our class to *any* type, e.g., Double

```
    @Override
    public boolean equals(Object o) {
        if(o == this) return true;                  // An object is equal to itself
        if(o == null) return false;                 // Null objects equal nothing
        if(o instanceof Double)
            return (a == (Double) o) && b == 0;     // Compare Complex to Double
        if(getClass() != o.getClass()) return false; // Different type is not equal
        Complex c = (Complex)o;                     // Create a Complex reference
        return (a == c.a) && (b == c.b);            // Compare two Complex by fields
    }
```

(Again, this code is incomplete but illustrates the principle)

```
public class TestComplex {
    public static void main(String[] args) {
        Complex c3 = new Complex(5.0, 0.0);
        Double d = 5.0;
        System.out.println("" + c3 + " "
            + ((c3.equals(d)) ? " equals " : " does not equal ")
            + " " + d);
    }
```

```
ricegf@antares:~/dev/202108/04/code_from_slides/complex04$ javac TestComplex.java
ricegf@antares:~/dev/202108/04/code_from_slides/complex04$ java TestComplex
5.0+0.0i  equals  5.0
ricegf@antares:~/dev/202108/04/code_from_slides/complex04$
```

Primitives vs Objects

- Primitives – int, double, char, boolean – have no methods because they are *primitive*
  - Their value is their *actual* value rather than their address
  - If on the stack or in a register, they HAVE no address!
- For doubles x and y, **`x == y`** is fine (no *object*, no equals *method*!)
- But double isn't an Object – for that we have Double
  - Double is the object form of double (**`Double d = 3f;`**), as Integer is for int (**`Integer I = 42;`**)
  - "Autoboxing" converts a primitive to its Object form when needed, so that **`d.equals(5.0)`** will also work: 5.0, a double, is auto-converted to Double so that **`d.equals(new Double(5.0))`** is thankfully NOT needed!

Dangerous Doubles

- Note that comparing doubles is a special case in *any* language
  - We can't represent infinite double precision with finite bits, so tiny differences can give false negatives

```
public class Roundoff {
    public static void main(String[] args) {
        double d1 = 0.1;
        double d2 = 0.2;
        double result = d1 + d2;
        System.out.println(d1 + " + " + d2
            + ((result == 0.3) ? " == " : " != ") + "0.3");
        System.out.println("0.1 + 0.2  = " + result);
    }
}
```

```
ricegf@antares:~/dev$ javac Roundoff.java
ricegf@antares:~/dev$ java Roundoff
0.1 + 0.2 != 0.3
0.1 + 0.2  = 0.30000000000000004
ricegf@antares:~/dev$
```

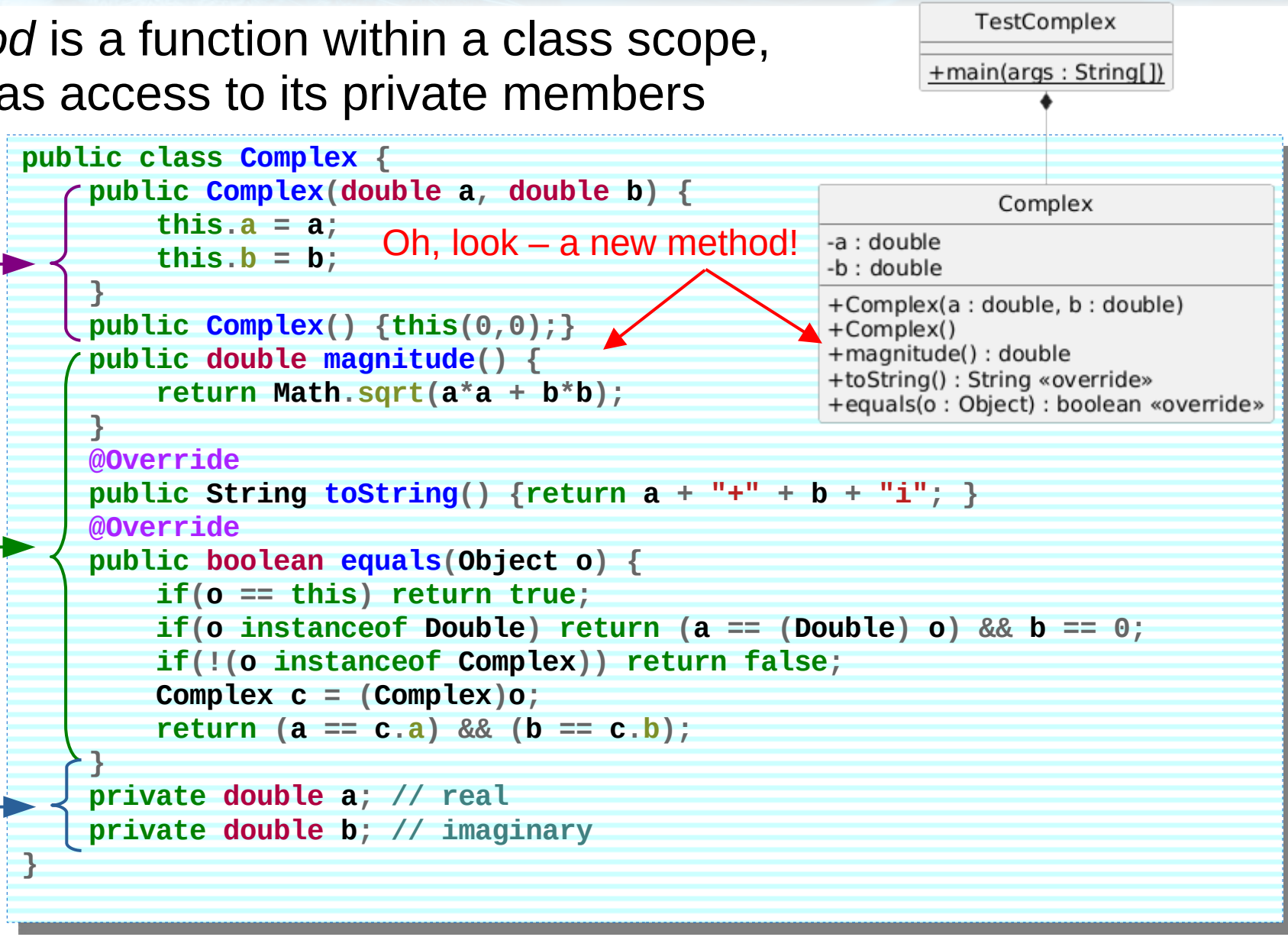
```
        double epsilon = 0.000001;
        System.out.println(d1 + " + " + d2
            + ((Math.abs(result - 0.3) < epsilon) ? " == " : " != ") + "0.3");
```

```
ricegf@antares:~/dev$ java RoundoffFix
0.1 + 0.2 == 0.3
0.1 + 0.2  = 0.30000000000000004
ricegf@antares:~/dev$
```

Compare the absolute difference to "epsilon" (a very small double)

Methods
(Directory complex05)

- A *method* is a function within a class scope, which has access to its private members




```
public class Complex {
    public Complex(double a, double b) {
        this.a = a;
        this.b = b;
    }
    public Complex() {this(0,0);}
    public double magnitude() {
        return Math.sqrt(a*a + b*b);
    }
    @Override
    public String toString() {return a + "+" + b + "i"; }
    @Override
    public boolean equals(Object o) {
        if(o == this) return true;
        if(o instanceof Double) return (a == (Double) o) && b == 0;
        if(!(o instanceof Complex)) return false;
        Complex c = (Complex)o;
        return (a == c.a) && (b == c.b);
    }
    private double a; // real
    private double b; // imaginary
}
```

Instance vs Method

- A method* can only be called on an *instance* of the class

```
public class TestComplex {
    public static void main(String[] args) {
        Complex c1 = new Complex(3.0, 4.0);
        Complex c2 = new Complex(3.0, 4.0);
        Complex c3 = new Complex(5.0, 0.0);
        Complex c4 = new Complex(0.0, 3.14);
        System.out.println("The magnitude of " + c1 + " is " + c1.magnitude());
        System.out.println("The magnitude of " + c3 + " is " + c3.magnitude());
        System.out.println("The magnitude of " + c4 + " is " + c4.magnitude());
    }
}
```

Instance (object)

Method

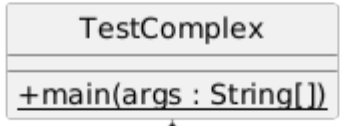
This is similar to passing a struct to a function in C, for example, `magnitude(c4)`

```
ricegf@antares:~/dev/202108/04/code_from_slides/complex05$ javac TestComplex.java
ricegf@antares:~/dev/202108/04/code_from_slides/complex05$ java TestComplex
The magnitude of 3.0+4.0i is 5.0
The magnitude of 5.0+0.0i is 5.0
The magnitude of 0.0+3.14i is 3.14
ricegf@antares:~/dev/202108/04/code_from_slides/complex05$
```

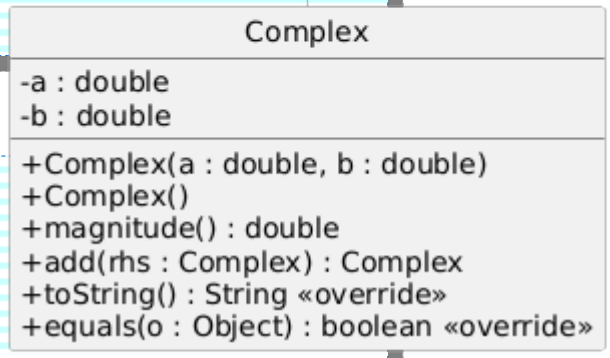
* Technically a *non-static* method. We'll get to this distinction next.

The Add Method
(Directory complex06)

- Complex numbers add by adding the real and imaginary portions, respectively




```
    public Complex add(Complex rhs) { // rhs is "right-hand side" of the +
        return new Complex(a+rhs.a, b+rhs.b);
    }
```




```
public class TestComplex {
    public static void main(String[] args) {
        Complex c1 = new Complex(3.0, 4.0);
        Complex c3 = new Complex(5.0, 0.0);
        Complex c4 = new Complex(0.0, 3.14);
        Complex c5 = c1.add(c3).add(c4); // c5 = c1 + c3 + c4
        System.out.println("" + c1 + " + " + c3 + " + " + c4 + " = " + c5);
    }
}
```

```
ricegf@antares:~/dev/202108/04/code_from_slides/complex06$ javac TestComplex.java
ricegf@antares:~/dev/202108/04/code_from_slides/complex06$ java TestComplex
3.0+4.0i + 5.0+0.0i + 0.0+3.14i = 8.0+7.140000000000001i
ricegf@antares:~/dev/202108/04/code_from_slides/complex06$
```

Look – round-off error!

Chaining Methods

- By returning the result, we can *chain* the add operations
  - Java permits methods to be called on objects returned by other methods
  - This is another form of chaining in Java, and *very* common!
- Compare the method approach to primitive operators
  - `Complex c5 = c1.add(c3).add(c4); // Java`
  - `Complex c5 = c1 + c3 + c4; // C++`
  - Your preference may vary
  - (In some languages such as C++ and Python, you are permitted to define operators such as + for your own types. Java does NOT permit this. We'll show you how to write operators in C++.)

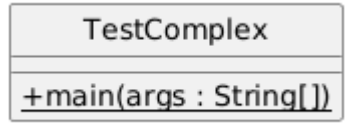
Static Class Members

- A static method or field exists as part of the *class*, and its *one* memory location is shared among *all* instances
  - A **static method** may be called without instancing an object, but cannot access any non-static members of the class
  - A **static field** (variable) is usually initialized in-line, not via the constructor

```
public class Complex {
    public enum Form{CARTESIAN, POLAR}; // and EXPONENTIAL, but who's counting?

    // Form to use for I/O (constructor and toString)
    private static Form form = Form.CARTESIAN;

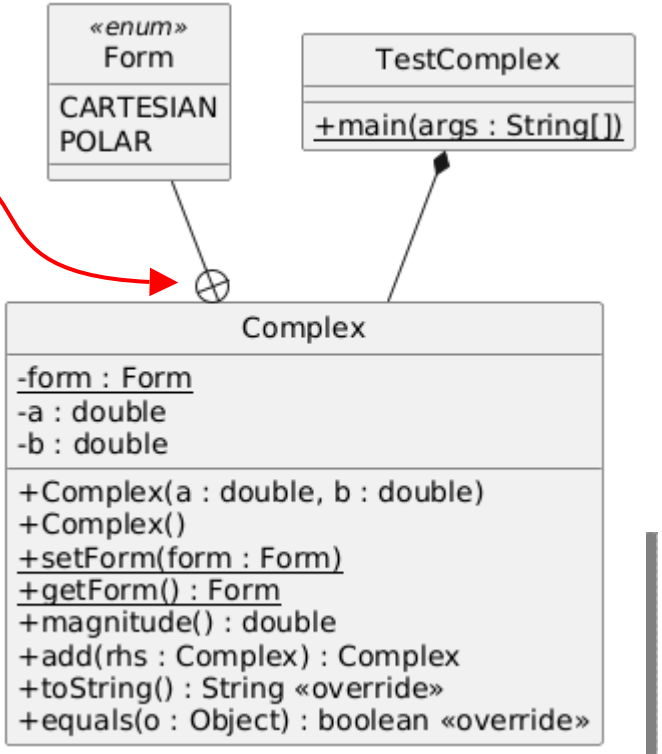
    public static void setForm(Form form) {
        Complex.form = form;
    }
    public static Form getForm() {
        return form;
    }
    // To be continued...
```



Note: In the UML, static members are underlined

Static Class Members

The ⊕ symbol represents “nesting”.
Note that enum Form is declared
as part of (nested within) class Complex.




```
public class Complex {
    public enum Form{CARTESIAN, POLAR};

    // Form to use for I/O (constructor and toString)
    private static Form form = Form.CARTESIAN;

    public static void setForm(Form form) {
        Complex.form = form;
    }
    public static Form getForm() {
        return form;
    }
    // To be continued...
```

Handling Static Fields

- One memory location is allocated for a static field, shared by ALL objects
  - Thus, a static field is “global” to the class
  - Good for handling characteristics and properties of the class, such as whether to globally use Cartesian or polar notation
- The static field can be accessed via the class name
  - For example, **`Complex.form`** is valid even if class Complex has never been instanced!
- The static field may also be accessed via an object
  - Every object accesses the same static field – same memory address
  - Static method **`Complex.setForm`** can access static field **`form`**, but NOT non-static fields **`a`** or **`b`**
    - Because if the class hasn’t been instanced, neither **`a`** nor **`b`** (nor **`this`**) exists!

Changing Behavior on a Static Field

- Static fields can modify all objects' behaviors with a single change
  - In this case, **`boolean form`** determines if the constructor parameters are Cartesian or polar

```
public Complex(double a, double b) {
    switch (form) {
        case CARTESIAN: {
            this.a = a;
            this.b = b;
            break;
        }
        case POLAR: {
            this.a = a * Math.cos(b);
            this.b = a * Math.sin(b);
            break;
        }
        default: throw new IllegalArgumentException("Invalid Form enum");
    }
}
```

Eventually someone will add **`EXPONENTIAL`** – will they remember this switch?

Changing Behavior on a Static Field

- **`boolean form`** also specifies the output format as Cartesian or polar

```
@Override
public String toString() {
    switch (form) {
        case CARTESIAN: {
            return a + "+" + b + "i";
        }
        case POLAR: {
            final double r = Math.sqrt(a*a + b*b);
            final double theta = Math.atan(b/a);
            return r + "(cos " + theta + ") + i sin(" + theta + ")";
        }
        default: throw new IllegalArgumentException("Invalid Form enum");
    }
}
```

- For formatted output, try

```
String r = String.format("%.2f", Math.sqrt(a*a + b*b));
String theta = String.format("%.3f", Math.atan(b/a));
```

Using Static Members

- Notice how the Form enum elements are specified
- Complex.setForm is also called using c2.setForm – by class or object

```
public class TestComplex {
    public static void main(String[] args) {
        Complex c1 = new Complex(3.0, 4.0);         // Cartesian by default - a + bi
        Complex.setForm(Complex.Form.POLAR);        // Call static method on CLASS
        Complex c2 = new Complex(10.0, Math.PI/4); // Polar - r(cos θ + i sin θ)

        System.out.println("In polar:     " + c1 + " and\n              " + c2);

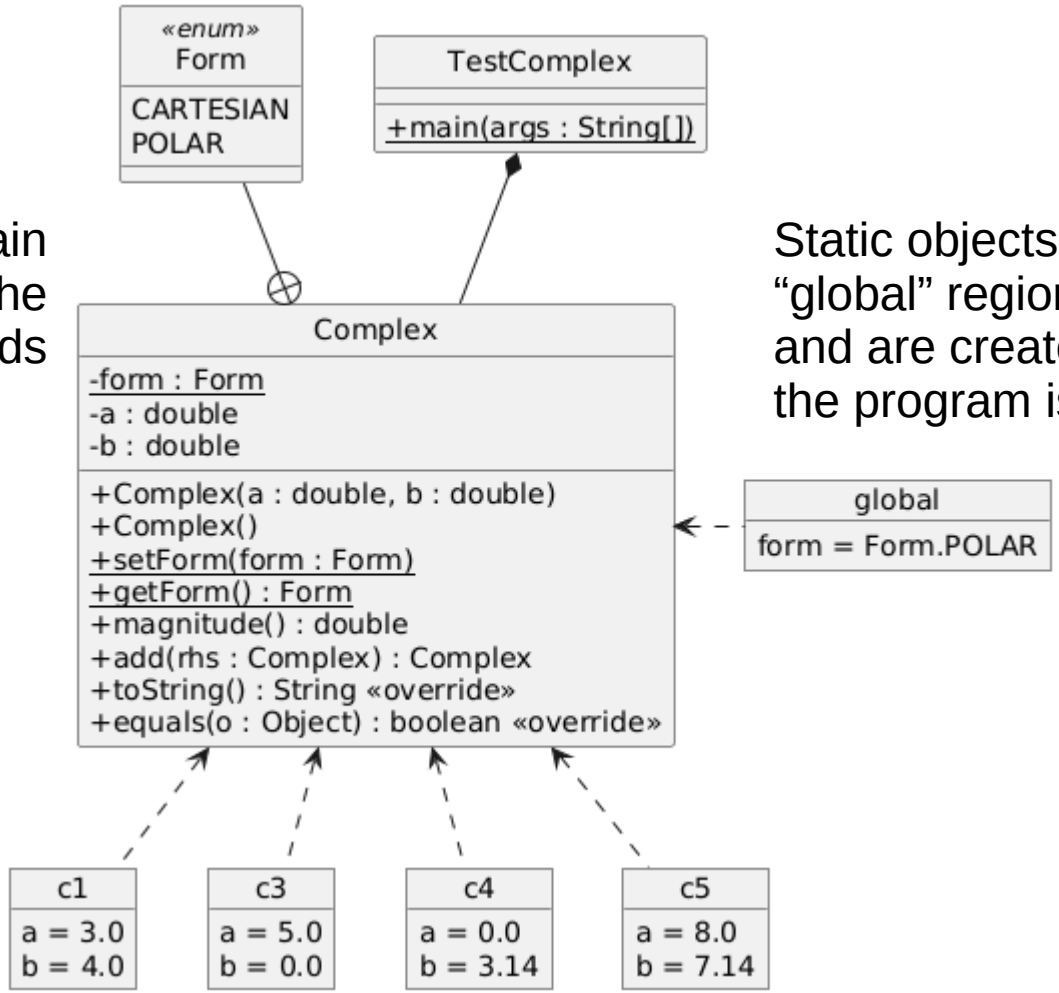
        c2.setForm(Complex.Form.CARTESIAN);         // Call static method on OBJECT

        System.out.println("In Cartesian: " + c1 + " and\n              " + c2);
    }
}
```

```
ricegf@antares:~/dev/202108/04/code_from_slides/complex07$ javac TestComplex.java
ricegf@antares:~/dev/202108/04/code_from_slides/complex07$ java TestComplex
In polar:     5.0(cos 0.9272952180016122) + i sin(0.9272952180016122) and
              10.0(cos 0.7853981633974483) + i sin(0.7853981633974483)
In Cartesian: 3.0+4.0i and
              7.071067811865475+7.071067811865475i
ricegf@antares:~/dev/202108/04/code_from_slides/complex07$
```

Class vs Object
Static vs Non-Static Fields

The class and enum contain the method code and the template for the fields

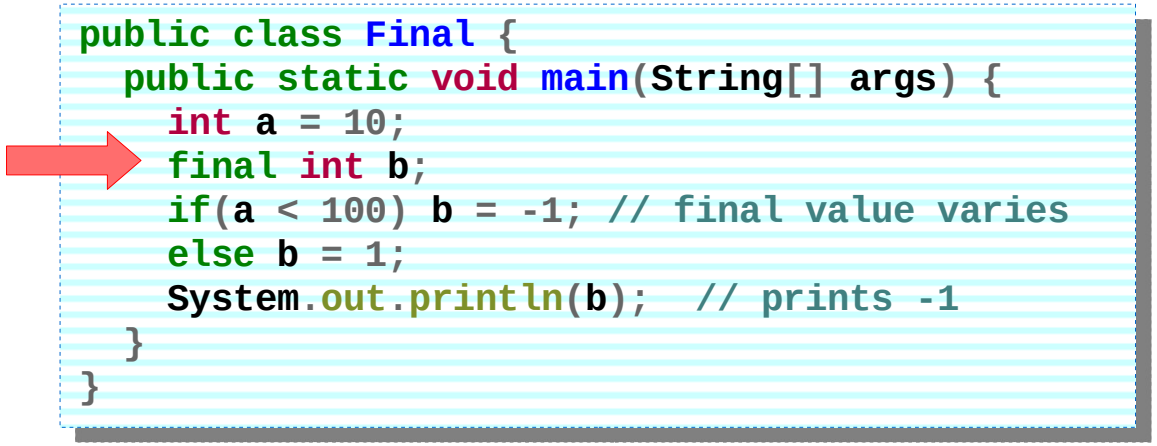


Static objects exist in a "global" region of memory and are created when the program is loaded

The NON-static objects simply fill in the fields template from the class

Final for Fields*

- Java's **`final`** keyword is similar to C/C++ **`const`**
  - **`final int a = 10;`** means **`a`**'s value is always 10
  - **`final int b;`** as a field means the value of **`b`** may be initialized exactly once in the constructor, otherwise it will be fixed at its default value
  - **`final int b;`** in a method means **`b`** can subsequently be initialized exactly once, for example,



```
public class Final {
  public static void main(String[] args) {
    int a = 10;
    final int b;
    if(a < 100) b = -1; // final value varies
    else b = 1;
    System.out.println(b);  // prints -1
  }
}
```

* We'll cover **`final`** for methods and classes in Lecture 07

Separate Compilation in Java

- Javac follows these rules when it finds a reference to class B while compiling class A
  - It checks for B.class in the current directory, and uses it if found *and if newer than B.java* (if B.java exists). If not,
  - It checks for B.java in the current directory, and compiles and uses it if found. If not,
  - It checks for B.class in the Java library*, and uses it if found. If not,
  - It reports that it cannot find class B and aborts
- Simply compiling the class that contains your main method will also compile the classes it uses
  - But **re**compiling when needed isn't guaranteed, therefore
  - **ALWAYS include our standard ant build.xml with your homework**
  - Use **`ant clean ; ant`** when you want to be certain! Our TAs do...

* You can add a custom path on the command line using **`java -Djava.library.path="/path/to/my/library"`**

Building “Large” Projects

- Your *smaller* CSE1325 projects *could* be easily built with just the javac command (though Ant helps)
- Much *larger* projects need more automation than Ant
  - Apache Maven, for example, specifies how to efficiently build a large Java project
  - JUnit and TestNG, for example, support creating regression tests for a Java project of any size
  - Jenkins, for example, automates event-based builds, tests, and deployments, for example, “build on each merge to main branch, deploy to test server, and run all regression tests”
    - Jenkins supports “**continuous integration**”, or frequently (more than once per day) merging all developers’ updates into a main development branch
- We will not discuss these further

“Intellectual Property”

In the United States (Other Nations May Vary)

- “To promote the Progress of Science and useful Arts, by securing for limited Times to Authors and Inventors the exclusive Right to their respective Writings and Discoveries”

  – US Constitution, Article 1, Section 8, ¶ 8

- Three primary types of interest to CSE
  - **Trademark** – symbol or name established by use as representing a company or product
  - **Patent** – exclusive right to make, use, or sell an invention
  - **Copyright** – exclusive right to print, publish, perform, execute, or record a creative work or its derivatives, and to authorize others to do the same

I am NOT a Lawyer!

The information in this lecture is for educational purposes only and not for the purpose of providing legal advice. You should contact a competent attorney to obtain specific advice with respect to any actual legal issue.

Scales by Visualityswiss per the Pixabay License https://pixabay.com/illustrations/justice-law-libra-attorney-2071621/

Trademark

- Trademarks avoid customer confusion
  - “Microsoft Windows” is a specific operating system
  - It is illegal to sell another OS product by that name
- Established simply by use, but better to register*
  - Public notice that it's your trademark
  - Stronger court case against infringement
- Lasts as long as it is defended
  - Xerox used to be a trademark for copying papers

* Register via the US Patent and Trademark Office, or USPTO

http://www.uspto.gov/sites/default/files/trademarks/basics/BasicFacts.pdf

Patent

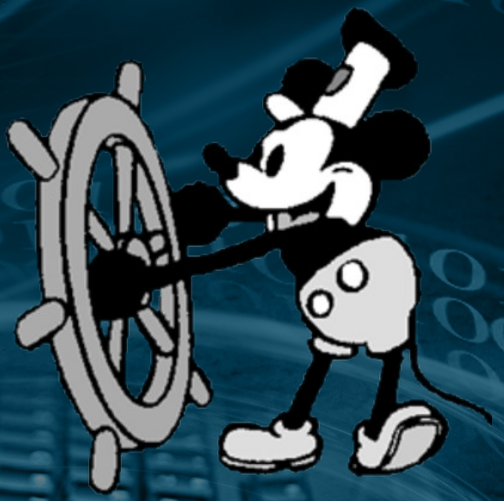
- Patents protect new or improved processes, machines, manufactured articles, and states of matter if they are:
  - Novel and non-obvious
  - Not yet published or in general use
  - Not previously patented
- Patents must be filed with the USPTO, and generally last for 20 years from date of application
  - “Patent pending” gives notice of a patent application

http://www.uspto.gov/patents-getting-started/general-information-concerning-patents

Copyright

- Copyright protects creative works
  - Copyright is automatic on creating the new work
  - Registering the copyright offers better protection
- Copyright* now lasts the shorter of:
  - 70 years after the death of the last surviving author
  - 95 years after publication
  - 120 years after creation

"for limited Times"?



In the Public Domain at last!
(First expected in 1984!)

- CSE who work for corporations usually sign a "work for hire" contract, assigning copyright to the employer
  - Even for non-work related software you write
  - You can – and probably should – negotiate limited exclusions

http://www.uspto.gov/learning-and-resources/ip-policy/copyright/copyright-basics

* Works created prior to 1978 follow different rules

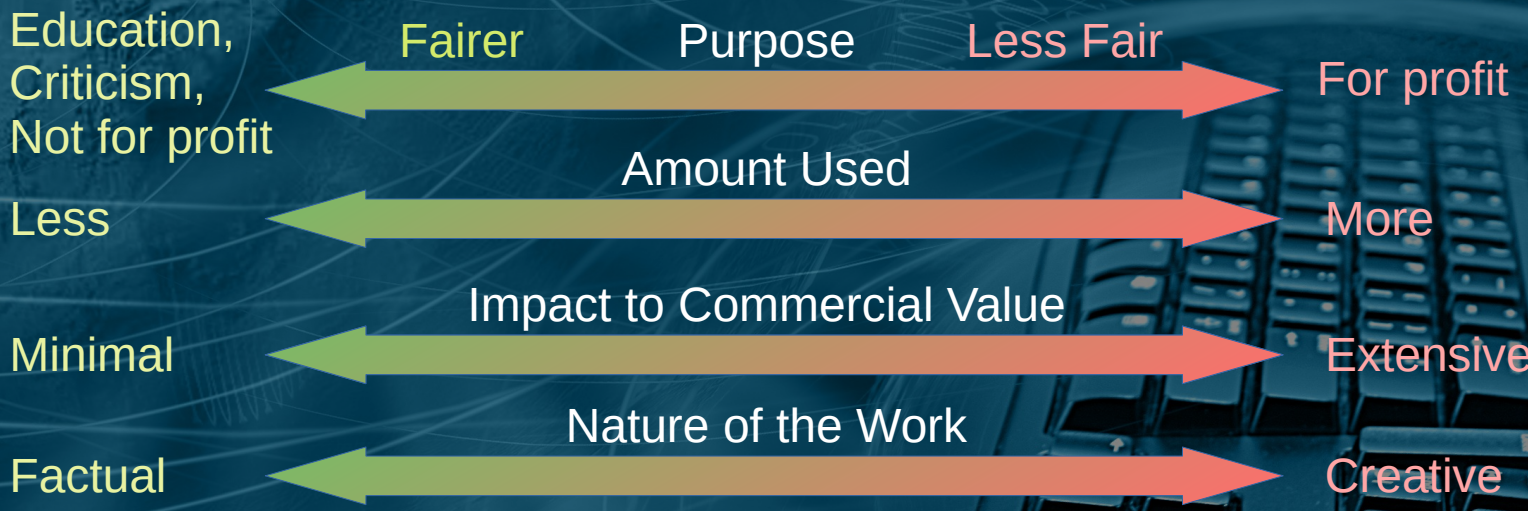
Types of Software Licenses

- If no license is specified, then “all rights reserved”
  - **Public Domain** - all ownership is disclaimed (SQLite)
  - **Permissive** (MIT, BSD, Apache) permits use, copying, distribution, and (usually with attribution) derivatives, even proprietary derivatives (BSD Unix, Apache)
  - **Protective** (GPL 2, 3, Lesser GPL, EPL) permits use, copying, distribution, and derivatives *with share-alike rules* (Linux, git)
    - GPL 3 also includes important patent clauses
  - **Shareware** permits (sometimes limited) use, copying, and (usually) distribution (Irfanview, early WinZip releases)
  - **Proprietary** permits (often restricted) use (Windows, Photoshop)
  - **Trade Secret** typically restricts (knowledge of and) use to the copyright holder

http://www.gnu.org/licenses/license-list.en.html

Affirmative Defense: Fair Use

- You (as I do for this class) may use US-copyrighted material *without* a license as “fair use” in certain *limited* contexts
  - “Fairness” is valuated at trial according to 4 criteria:

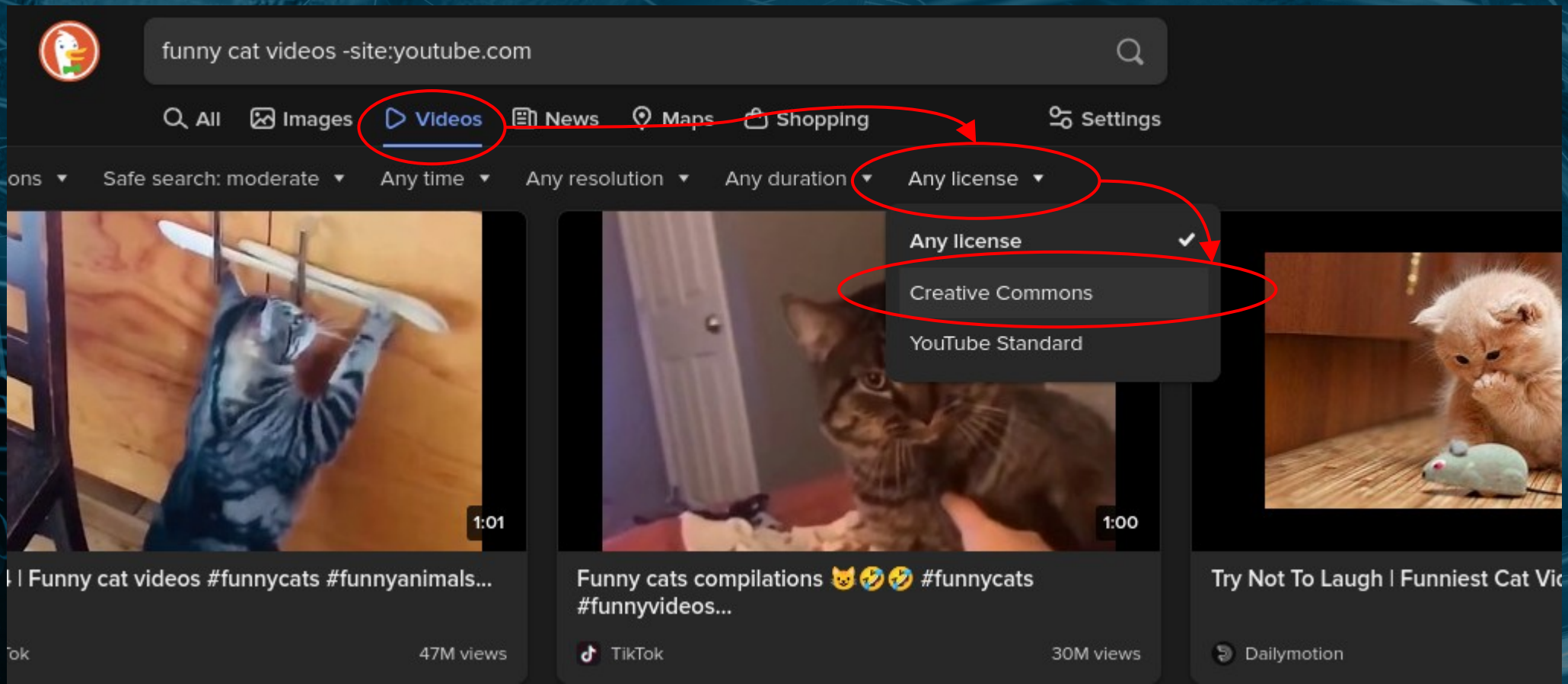


- The DMCA specifies “take down notices” by which social media deletes allegedly infringing posts
  - “Fair Use” must be considered when a take down is appealed

* Digital Millennium Copyright Act of 1998

Finding Material to Legally Reuse

- Search, for example, using https://duckduckgo.com/, https://images.google.com/ , or https://videos.search.yahoo.com/ and enter search term(s)
- Click License, Filter, or Tools and select a License type
- Select the resource and *verify the license on the website*



Common Free Culture Websites

- These are some resources I frequently use
  - https://pixabay.com (videos, photos, and illustrations)
  - https://commons.wikimedia.org (Wikipedia's images)
  - https://thenounproject.com (photos and icons)
  - https://freepik.com (mixed free / commercial icons)
  - https://flaticon.com (mixed free / commercial icons)
  - https://ccMixter.com (free audio mixes)
- Mentions here do not imply endorsement
- Licenses may vary
  - ALWAYS **check** and **conform to** the license

Note: YouTube is determined to NOT let you reuse / remix hosted videos, even those explicitly in the public domain or under Creative Commons licenses

Software License Significance

Licensed under Creative Commons Zero – CC0 (public domain)
https://www.pikrepo.com/fbxsm/man-in-white-dress-shirt-holding-magnifying-glass

Licensed under the Pexels License, similar to CC0
https://www.pexels.com/photo/button-career-click-click-on-357866/

- Know the license for the software and resources you use & its requirements
  - If you have questions, ask an expert
  - If you can't follow them, don't use that software
- Select and Document a license for *everything* you author
  - If a “work for hire”, ask your manager
  - If an independent work, choose wisely

Always follow copyright law – it's your job