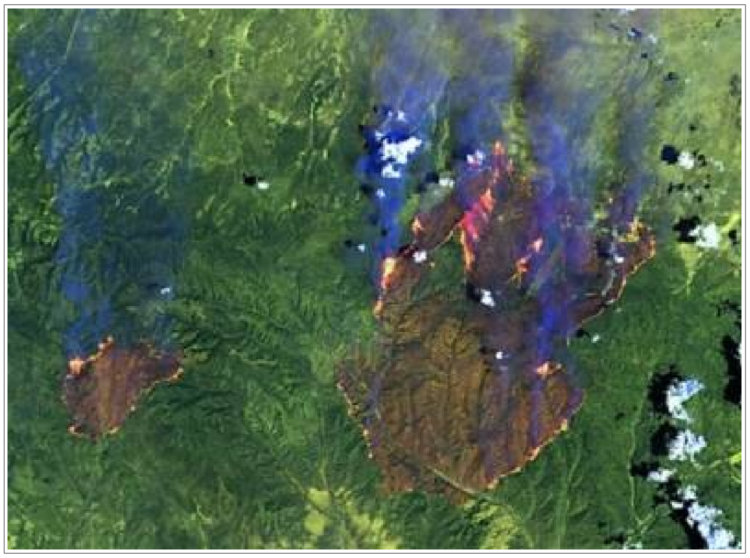
<< Lecture 10 and 11 >>

Preamble

- Grand topic:
  - **Object orientation**
    - Principles
    - Variables and identity
    - **Polymorphism**

Homework 2

- Two problems—polymorphism and encapsulation
- Problem 1: easy, fast
- Problem 2: **forest fires**
  - Challenging—meant to raise your skills
  - Actual basic simulation (discrete time/step)
  - Object state, polymorphism, encapsulation
- The homework comes with methods to [partially] test your code!



Design goals of OOP

1. Robustness: *the characteristic of the software artifact to tolerate or handle critical errors*
   - OO languages provide facilities for dealing with exceptional behavior
2. Adaptability: *the characteristic of the software artifact that permits its use or easy adaption to different scenarios*
   - A.k.a. evolvability, and related to portability and reusability, adaptability is about
3. Reusability: *the characteristic of the software artifact that allows its easy re-use in new projects*
   - With a well-defined, clean interaction with the rest of the system, the code of the *modules* can be re-used in new projects, shortening development and testing time

Principles of OOP

1. Abstraction: *to simplify a system to its most fundamental parts or ideas*
    - Our code is always an abstraction anyways, because our thought are; when we think of trees, we are thinking about the atoms, etc., nor we have trees physically in our brains, but only representations of them
    - Abstraction means, however, to reduce *the concepts* to just the *necessary* degree of complexity for effectively making use of them in the software code
    - Abstraction calls for *simplicity* and *generality*
2. Encapsulation: *that software components should not need to reveal internal details*
    - As we have various facilities provided by in Java, such as arrays, the String class, the System.out stream, etc., we should be aware that we can use these facilities without the need of understanding their internals
3. Modularity: *that software components should each one represent one well-defined functionality*
    - For example, methods in classes String, Boolean, Integer, and Character, were only pertaining to text, *boolean*s, *int*s, and *char*s, respectively
    - Modularity calls for the software artifact to be properly decomposed in *coherent* subparts, making it easy to work with them
        - A coherent organization of the code of the software artifact makes it easy to remember where the available functionalities are located

Reflection questions

To properly grasp these concepts demands reflection. They also lead to *vague* questions in exams (can't be otherwise).

1. Provide an example of abstraction from mathematics
2. How do you experience encapsulation when programming in Java? Describe a situation
3. Match the OOP principles to its design goals (might appear in exams)
4. Give examples of life critical software

Object identity

- It is often thought as a core concept in OOP, but it is not always listed as one
- For objects, variables are names that *point* to memory addresses, where objects reside
- Many variables can point to the same object
- Does not happen with primitive types—variables store values rather than memory addresses

State of an object

- Objects often have *fields* (again, fields are variables defined outside of methods)
- Fields represent the internal state of the object
- If the values of the fields change, then the object is said to have *mutated*
  - Such objects are *mutable*
  - Strings are *immutable*, arrays are *mutable*
- Example application: you can "*return*" through the arguments of a method if the arguments are *mutable*
- Example code: "squareIt"

```
import java.util.Arrays;

public class Demo{

 // this methods squares the components of the supplied array
 private static void squareIt(int[] x) {
   for(int i=0; i<x.length; i++)
     x[i] = x[i]*x[i];   // mutation!
 }

 // main class
 public static void main(String[] argv) {
   int[] intarr = {0, 1, -1, 2, -2, 3, -3};
   System.out.println( Arrays.toString(intarr) );  // quickly prints the array
   squareIt( intarr );
   System.out.println( Arrays.toString(intarr) );
   squareIt( intarr );
   System.out.println( Arrays.toString(intarr) );
 }

}
```

For the above code, Netbeans should show the following upon execution:

```
run:
[0, 1, -1, 2, -2, 3, -3]
[0, 1, 1, 4, 4, 9, 9]
[0, 1, 1, 16, 16, 81, 81]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Polymorphism

- *Poly* means many, and *morph* means shape—Polymorphism: that types can assume multiple types
- In object oriented languages, types can often assume a family of other types through the process of subtyping (e.g., inheritance)
- Remember casting? Casting assumes renewed importance now with *compatible* types

- Types of polymorphism and Java:
    - Ad-hoc polymorphism: method overloading
    - Subtyping: today's class
    - Parametric typing: soon

Polymorphism

- *Poly*: many; *morph*: shape.
- Polymorphism: that types can assume other types
- Remember casting?
- Polymorphism in Java:
    - *Ad-hoc polymorphism*: that methods change their behavior based on the different types they are supplied
        - Java provides facilities for ad-hoc polymorphism through *method overloading*
    - *Subtyping*: that types can have one or more supertypes associates
        - In subtyping, the supertype is not always compatible with the subtype, but the subtype is always compatible with the supertype
        - Java provides two facilities for this: *extension* (inheritance) and *implementation* (interfaces)
        - **Subtypes inherit all the types of the supertypes**
    - *Parametric typing*: that types can be defined by having a parameter
        - Java provides the mechanism of *generics* to address parametric typing, much alike the mechanism of *templates* in C++
        - Methods can also have parametric types
            - This is not ad-hoc polymorphism however, as we will see

Example: animals

- Suppose we have a real world simulation of an ecosystem, where animals of one kind feed on another
- Some objects of the ecosystem will be animals
- Animals come in many subtypes:
    - Mammals
        - dogs, cats, monkeys, bats, whales, horses, ...
    - Birds

        - sparrows, hawks, ducks, parrots, penguins, vultures, ...
    - Reptiles
        - iguanas, lizards, snakes, crocodiles, ...
    - Fish, etc...
    - In general, we will organize the categories as:
        - Animal > *Kingdom* > *Species*

<<dogs in their ecosystem>>

<<they might be licking each other, but they are still competing for the shoe>>

- Also, some kinds of animals might use the lands, others the waters, others the skies
- How can we represent this? We might want to use **types**
    - Animals will be **supertype** here
    - Mammals is a **subtype** of Animals
    - Dogs is a **subtype** of Mammals
    - (We might want to use even more types in between)
- Example: **extends**
- It is possible to make classes **abstract:** no *instantiation* (no *new*)
- Multiple inheritance: Java does not allow proper multiple inheritance
    - **Multiple subtyping**: **implements**—Java allows multiple subtyping by using the **interface** class definition

<< code examples in the next notes >>