CS 1331 Introduction to Object Oriented Programming

Data Abstraction

Christopher Simpkins
chris.simpkins@gatech.edu

Data Abstraction

An abstraction of a concept attempts to capture the essence of the concept – its essential properties and behaviors – by ignoring irrelevant details.

- Process abstraction - group the operations of a process in a subprogram and expose only the essential elements of the process to clients through the subprogram signature (e.g., function/method name and parameters)
- Data abstraction - encapsulation of data with the operations defined on the data
- A particular data abstraction is called an *abstract data type*. Note that ADT's include process abstractions as well

In each case, an abstraction hides details — details of a process or details of a data structure.

> *"Abstraction is selective ignorance."*
> *– Andrew Koenig (C++ Guru)*

A Complex Number ADT

ADT: Complex
Data:

- **real: double** the real part of a complex number
- **imaginary: double** the imaginary part of a complex number

Operations:

- **new** - construct a new complex number
- **plus** - add one complex number to another, yielding a new complex number

An ADT is *abstract* becuase the data and operations of the ADT are defined independently of how they are implemented. We say that an ADT *encapsulates* the data and the operations on the data.

Data Abstractions with Classes

Java provides langauge suppport for defining ADTs in the form of classes.
A class is a blueprint for objects. A class definition contains

- instance variables, a.k.a. member variables or fields – the state, or data of an object
- methods, a.k.a. member functions or messages – the operations defined on objects of the class

We *instantiate* or *construct* an object from a class.

Java Implementation of Complex Number ADT

Here's a Java implementation of our complex number ADT [1]:

```
public class Complex {
    // These are the data of the ADT

    private double real;
    private double imaginary;

    // These are the operations of the ADT

    public Complex(double aReal, double anImaginary) {
        real = aReal;
        imaginary = anImaginary;
    }

    public Complex plus(Complex other) {
        double resultReal = this.real + other.real;
        double resultImaginary = this.imaginary + other.imaginary;
        return new Complex(resultReal, resultImaginary);
    }
}
```

[1] http://introcs.cs.princeton.edu/java/33design/

Reference Variables

Consider the following code:

```
Complex a = new Complex(1.0, 2.0);
Complex b = new Complex(3.0, 4.0);
Complex c = a.plus(b);
```

`a`, `b`, and `c` are *reference* variables of type `Complex`. Reference variables have one of two values:

- the address of an object in memory (in this case an instance of `Complex`), or
- `null`, meaning the variable references nothing.

Invoking Constructors

The line:

```
Complex a = new Complex(1.0, 2.0);
```

invokes the `Complex` constructor, passing arguments `1.0` and `2.0`:

```
public Complex(aReal= 1.0 , anImaginary= 2.0 ) {
    real = 1.0 ;
    imaginary = 2.0 ;
}
```

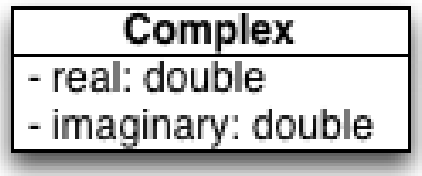
which *instantiates* a `Complex` object and stores its address in the variable `a`:

```
Complex a = new Complex(1.0, 2.0);
```

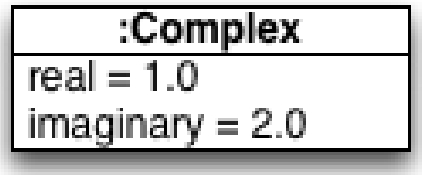
Constructors initialize objects. After the line above, `Complex` object `a`'s instance variables have the values `1.0` and `2.0`.

Visualizing Objects and Instantiation

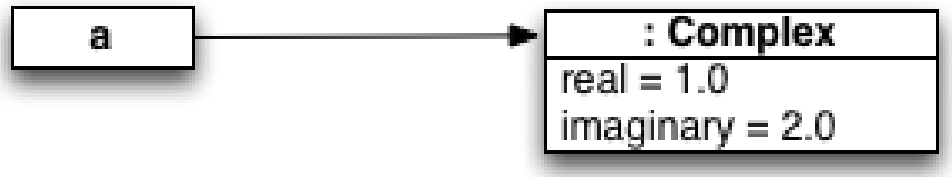
The object creation expression `new Complex(1.0, 2.0)` applies the `Complex` blueprint defined by the class definition from slide 5:



to the constructor arguments `(1.0, 2.0)` to create an instance of `Complex`:



We can assign this object to a reference variable, e.g., `Complex a = new Complex(1.0, 2.0)`:



Invoking Methods on Objects

The line:

```
Complex c = a.plus(b);
```

invokes the `plus` method on the `a` object, passing the `b` object as an argument, which binds the object referenced by `b` to the parameter `other`:

```
a.plus(other= b ) {
  double resultReal = this.real + b .real; // 1.0 + 3.0
  double resultImaginary = this.imaginary + b .imaginary; // 2.0 + 4.0
  return new Complex(resultReal, resultImaginary);
}
```

which returns a new `Complex` object and assigns its address to the reference variable `c`.

Using the `Complex` Class

Users, or *clients* of the `Complex` class can then write code like this:

```
Complex a = new Complex(1.0, 2.0);
Complex b = new Complex(3.0, 4.0);
Complex c = a.plus(b);
```

without being concerned with `Complex`'s implementation (which could use polar form, for example). Clients (i.e., users) of the `Complex` class need only be concerned with its interface, or *API* (application programmer interface) – the public methods of the class.

After the code above we have the following `Complex` objects in memory: