Lecture 1: Abstraction and Encapsulation

Learning Objectives

After this lecture, students should:

- recap some fundamental programming concepts, including the execution model of a program, abstractions over code and data, primitive and composite data types.
- appreciate the importance of maintaining abstraction barrier in software development
- understand the differences between statically and dynamically typed languages
- understand the concepts of object-oriented programming, including encapsulation, data hiding, fields and methods, constructors, mutators/accessors, classes and objects, and their purposes of introducing them as a method of programming.
- know the purpose and usage of Java keywords `class`, `public`, `private`, `final`, `static`, `import`, `new`
- understand that Java is a type-safe language, in contrast to C
- be familiar with Java variable and primitive types

What Exactly is a *Program*?

A program is a set of instructions we issue to computers to manipulate data. A programming language is a formal language that helps programmers specify precisely what are the instructions we issue to computers, using code that are often made up of keywords, symbols, and names. Computers execute the instructions in their *processing units*, and store the instructions and data in their *memory* 1 [#fn:1]. The processing units recognize the instructions based on the specific patterns of bits and manipulate data as a sequence of bits. A programming language, however, is written at a higher level of *abstraction* (i.e., at a higher conceptual level), so that as a programmer, we only need to write a few lines of code to give complex instructions to the computer. A *compiler* or *interpreter* is responsible for translating these programs written in high level language to *assembly code* or *machine code*, i.e., bit patterns that the processing units can understand.

There are thousands of programming languages in existence. *C* is one of the languages that is a *low-level language* -- i.e., it provides a very thin layer of abstractions on top of machine code. On the other hand, languages such as *Python* and *JavaScript* are high-level languages. As an example, in C, you can directly manage memory allocation. In JavaScript and Python, you cannot.

Abstraction: Variable and Type

One of the important abstractions that is provided by a programming language is *variable*. Data are stored in some location in the computer memory. But we should not be referring to the memory location all the time. First, referring to something like `0xFA49130E` is not user-friendly; Second, the location may change. A *variable* is an abstraction that allows us to give a user-friendly name to a piece of data in memory. We use the *variable name* whenever we want to access the *value* in that location, and *pointer to the variable* or *reference to the variable* whenever we want to refer to the address of the location.

Let's think a bit more about how a sequence of bits is abstracted as data in a programming language. At the machine level, these bits are just, well, bits. We give the bits a *semantic* at the program level, e.g., we want to interpret the sequence of bits as numbers, letters, etc. E.g., the number (integer, to be exact) `65` and the letter `A` all share the same sequence of bits `0100 0001` but are interpreted differently and possibly manipulated differently.

The *type* of a variable tells the compiler or the interpreter how to interpret the variable and how to manipulate the variable.

For instance, supposed that in Python, if you have two variables `x` and `y` storing the values `4` and `5` respectively, if you `print x + y`, you would get `45` if `x` and `y` are strings; you would get `9` if `x` and `y` are integers; you would get an error if `4` is an integer and `5` is a string.

In the last instance above, you see that assigning a type to each variable helps to keep the program meaningful, as the operation `+` is not defined over an integer and a string in Python 2 [#fn:2].

Python is a *dynamically typed* language. The same variable can hold values of different types, and checking if the right type is used is done during the execution of the program. Note that, the type is associated with the *values*, and the type of the variable changes depending on the value it holds.

C, on the other hand, is a *statically typed* language. We need to *declare* every variable we use in the program and specify its type. A variable can only hold values of the same type as the type of the variable, so we can't assign, for instance, a string to a variable of type `int`. We check if the right type is used during the compilation of the program.

```
int x = 4; // ok
int y = "5"; // error
```

By annotating each variable with its type, the C compiler also knows how much memory space is needed to store a variable.

Abstraction: Functions

Another important abstraction provided by a programming language is *function* (or *procedure*). This abstraction allows programmers to group a set of instructions and give it a name. The named set of instructions may take one or more variables as input parameters, and return one or more values.

Like all other abstractions, defining functions allow us to think at a higher conceptual level. By composing functions at increasingly higher level of abstractions, we can build programs with increasing level of complexity.

Defining functions allow us to abstract away the implementation details from the caller. Once a function is defined, we can change the way the function is implemented without affecting the code that calls the function, as long as the semantic and the *interface* of the function remains the same.

Function, therefore, is a critical mechanism for achieving *separation of concerns* in a program. We separate the concerns about how a particular function is implemented, from the concerns about how the function is used to perform a higher-level task.

Defining functions also allow us to *reuse* code. We do not have to repeatedly write the same chunk of code if we group the sequence of code into a function -- then we just need to call the function to invoke this sequence of code every time we need it. If this chunk of code is properly written and debugged, then we can be pretty sure that everywhere the function is invoked, the code is correct 3 [#fn:3].

C is a *procedural language*. A C program consists of functions, with the `main()` function serves as the entry point to the program. Since C is a statically typed language, a C function has a return type, and each function parameter (or *argument*) has a type as well. (Note that this statement does not mean that a C function must return a *value*. If the function does not return a value, we define its return type as `void`.)

Recall that the bits representing the instructions are also stored in the computer memory in an area separated from the data. The instructions that belong to the same function are stored in adjacent memory locations. Just like we can refer to a variable using its memory address through its *reference* (or *pointer*), we can refer to a function using the memory address of the entry point to the function.

Abstraction: Composite Data Type

Just like functions allow programmers to group instructions, give it a name, and refer to it later, a *composite data type* allows programmers to group *primitive types* together, give it a name (a new type), and refer to it later. This is another powerful abstraction in programming languages that help us to think at a higher conceptual level without worrying about the details. Commonly used examples are mathematical objects such as complex numbers, 2D data points, multi-dimensional vectors, circles, etc, or every day objects such as a person, a product, etc.

Defining composite data type allows programmers to abstract away (and be separated from the concern of) how a complex data type is represented.

For instance, a circle on a 2D plane can be represented by the center (`x` , `y`) and its radius `r` , or it can be represented by the top left corner (`x` , `y`) and the width `w` of the bounding square.

In C, we build composite data type with `struct`. For example,

```
struct circle {
  float x, y; // (x,y) coordinate of the center.
  float r; // radius
}
```

Once we have the `struct` defined, we are not completely shielded from its representation, until we write a set of functions that operates on the `circle` composite type. For instance,

```
float circle_area(circle c) { ... };
bool  circle_contains_point(circle c, point p) { ... };
  :
```

Implementing these functions obviously requires the knowledge of how a circle is represented. Once the set of functions that operates on and manipulates circles is available, we can use *circle* type without worrying about the internal representation.

If we decide to change the representation of a circle, then only the set of functions that operates on a circle type need to be changed, but not the code that uses circles to do other things.

We can imagine an *abstraction barrier* between the code that uses a composite data type along with its associated set of functions, and the code that define the data type along with the implementation of the functions. Above the barrier, the concern is about using the composite data type to do useful things, while below the barrier, the concern is about how to represent and manipulate the composite data type.

While many of you are used to writing a program solo, in practice, you rarely write a program with contributions from a single person. The abstraction barrier separates the role of the programmer into two: (i) an *implementer*, which define that data type and provide the implementation, and (ii) a *client*, which uses the composite data type to perform a higher level task [4] [#fn:4]. Part of my aim in CS2030 is to switch your mindset into thinking in terms of these two roles. Note that the implementer and the client may very well be the same programmer.

Abstraction: Class and Object (or, Encapsulation)

We can further bundle the composite data type and its associated functions together in another abstraction, called a *class*. A class is a data type with a group of functions associated with it. We call the functions as *methods* and the data in the class as *fields* (or *members*, or *states*, or *attributes* [4] [#fn:4]). A well-designed class maintains the abstraction barrier, properly wraps the barrier around the internal representation and implementation, and exposes just the right *interface* for others to use.

Just like we can create variables of a given type, we can create *objects* of a given class. Objects are *instances* of a class, each allowing the same methods to be called, and each containing the same set of variables of the same types, but (possibly) storing different values.

Recall that programs written in a procedural language such as a C consists of functions, with a `main()` function as the entry point. A program written in an *object-oriented language* such as Java consists of classes, with one main class as the entry point. One can view a running object-oriented (or OO) program as something that instantiates objects of different classes and orchestrates their interactions with each other by calling each others' methods.

One could argue that an object-oriented way of writing programs is much more natural, as it mirrors our world more closely. If we look around us, we see objects all around us, and each object has certain properties, exhibit certain behavior, and they allow certain actions. We interact with the objects through their interfaces, and we rarely need to know the internals of the objects we used everyday (unless we try to repair it) [5] [#fn:5].

The concept of keeping all the data and functions operating on the data related to a composite data type together within an abstraction barrier is called *encapsulation*.

Breaking the Abstraction Barrier

In the ideal case, the code above the abstraction barrier would just call the provided interface to use the composite data type. There, however, may be cases where a programmer may intentionally or accidentally break the abstraction barrier.

Consider the case of implementing `circle` as a C `struct`. Suppose someone wants to move the center of the circle `c` to a new position (`x` , `y`), instead of implementing a function `circle_move_to(c, x, y)` (which would still keep the representation used under the barrier), the person wrote:

```
c.x = x;
c.y = y;
```

This code would still be correct, but the abstraction barrier is broken since we now make explicit assumption that there are two variables `x` and `y` inside the `circle` data type that corresponds to the center of the circle. If one day, we want to represent a circle differently, then we have to carefully change all the code that read and write these variables `x` and `y` and update them.

Breaking Python's Abstraction Barrier

Python tries to prevent *accidental* access to internal representation by having a convention of prefixing the internal variables with `_` (one underscore) or `__` (two underscores). This method, however, does not prevent a lazy programmer from directly accessing the variables and possibly planting a bug/error that will surface later.

Data Hiding

Many OO languages allow programmers to explicitly specify if a field or a method can be accessed from outside the abstraction barrier. Java, for instance, supports `private` and `public` access modifiers [5 [#fn:5]]. A field or a method that is declared as `private` cannot be accessed from outside the class, and can only be accessed within the class. On the other hand, as you can guess, a `public` field or method can be accessed, modified, or invoked from outside the class.

Such mechanism to protect the abstraction barrier from being broken is called *data hiding* or *information hiding*. This protection is enforced by the *compiler* at compile time.

Example: The Circle class

Let's put together the concepts of encapsulation and data hiding to define a `Circle` class in Java:

```java
/**
 * A Circle object encapsulates a circle on a 2D plane.
 */
class Circle {
  private double x;  // x-coordinate of the center
  private double y;  // y-coordinate of the center
  private double r;  // the length of the radius

  /**
   * Return the area of the circle.
   */
  public double getArea() {
    return 3.1415926 * r * r;
  }

  /**
   * Move the center of the circle to the new position (newX, newY)
   */
  public void moveTo(double newX, double newY) {
    x = newX;
    y = newY;
  }
}
```

Here, we define `x`, `y`, and `r` as three private fields inside the class `Circle`. Note that these fields are not accessible and modifiable outside of the class `Circle`, but they can be accessed and modified within `Circle` (inside the abstraction barrier), such as in the methods `getArea` and `moveTo`.

Constructors, Accessors, and Mutators

With data hiding, we completely isolate the internal representation of a class using an abstraction barrier. With no way for the user of the class to modify the fields directly, it is common for a class to provide methods to initialize and modify these internal fields (such as the `moveTo()` method above). A method that initializes an object is called a *constructor*, and a method that retrieves or modifies the properties of the object is called the *accessor* (or *getter*) or *mutator* (or *setter*).

A constructor method is a special method within the class. It cannot be called directly but is invoked automatically when an object is instantiated. In Java, a constructor method *has the same name as the class* and *has no return type*. A constructor can take in arguments just like other functions. The class `Circle` can have a constructor such as the following:

```java
class Circle {
    :
  /**
   * Create a circle centered on (centerX, centerY) with given radius
  */
  public Circle(double centerX, double centerY, double radius) {
    x = centerX;
    y = centerY;
    r = radius;
  }
    :
}
```

The use of accessor and mutator methods is a bit controversial. Suppose that we provide an accessor method and a mutator method for every private field, then we are actually exposing the internal representation, therefore breaking the encapsulation. For instance:

```java
class Circle {
    :

  public double getX() {
    return x;
  }

  public void setX(double newX) {
    x = newX;
  }

  public double getY() {
    return y;
  }

  public void setY(double newY) {
    y = newY:
  }

  public double getR() {
    return r;
  }

  public void setR(double newR) {
    r = newR;
  }
    :
}
```

The examples above are pretty pointless. If we need to know the internal and do something with it, we are doing it wrong. The right approach is to implement a method within the class that do whatever we want the class to do. For instance, suppose that we want to know the circumference of the circle `c` , one approach would be:

```
double circumference = 2*c.getR()*3.1415926;
```

where `c` is a `Circle` object.

A better approach would be to add a new method `getCircumference()` in the `Circle` class, and call it instead:

```
double circumference = c.getCircumference();
```

The better approach involves writing a few more lines of code to implement the method, but it keeps the encapsulation intact. If one fine day, the implementer of `Circle` decided to store the diameter of the circle instead of the radius, then only the implementer needs to change the implementation of `getCircumference` . The client does not have to change anything.

Constructor in Python and JavaScript
In Python, the constructor is the `__init__` method. In JavaScript, the constructor is simply called `constructor` .

Class Fields and Methods

Let's look at the implementation of `getArea()` above. We use the constant π but hardcoded it as 3.1415926. Hardcoding such a magic number is a *no-no* in terms of coding style. This constant can appear in more than one places. If we hardcode such a number and want to change its precision later, we would need to trace down and change every occurrence. Every time we need to use π, we have to remember or look up what is the precision that we use. Not only does this practice introduce more work, it is also likely to introduce bugs.

In C, we define π as a macro constant `M_PI` . But how should we do this in Java? This is where the ideal that a program consists of only objects with internal states that communicate with each other feel a bit constraining. The constant π is universal, and does not really belong to any object (the value of π is the same for every circle!). Another example: if we define a method `sqrt()` that computes the square root of a given number, this is a general function that is not associated with any object as well.

A solution to this is to associate these *global* values and functions with a *class* instead of with an *object*. For instance, Java predefines a `Math` [https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html] class [6] [#fn:7] that is populated with constants `PI` and `E` (for Euler's number e), along with a long list of mathematical functions. To associate a method or a field with a class in Java, we declare them with the `static` keyword. We can additionally add a keyword `final` to indicate that the value of the field will not change [7] [#fn:8].

```
class Math {
  :
  public static final double PI = 3.141592653589793;
  :
  :
}
```

We call these fields and methods that are associated with a class as *class fields* and *class methods*, and fields and methods that are associated with an object as *instance fields* and *instance methods*.

Class Fields and Methods in Python
Note that, in Python, any variable declared within a `class` block is a class field:

```
class Circle:
  x = 0
  y = 0
```

In the above example, `x` and `y` are class fields, not instance fields.

Example: The Circle class

Now, let revise our `Circle` class to improve the code and make it a little more complete:

```
import java.lang.Math;

/**
 * A Circle object encapsulates a circle on a 2D plane.
 */
class Circle {
  private double x;  // x-coordinate of the center
  private double y;  // y-coordinate of the center
  private double r;  // the length of the radius

  /**
   * Create a circle centered on (centerX, centerY) with given radius
   */
  public Circle(double centerX, double centerY, double radius) {
    x = centerX;
    y = centerY;
    r = radius;
  }

  /**
   * Return the area of the circle.
   */
  public double getArea() {
    return Math.PI*r*r;
  }

  /**
   * Return the circumference of the circle.
   */
  public double getCircumference() {
    return Math.PI*2*r;
  }

  /**
   * Move the center of the circle to the new position (newX, newY)
```

```
     */
    public void moveTo(double newX, double newY) {
      x = newX;
      y = newY;
    }

    /**
     * Return true if the given point (testX, testY) is within the circle.
     */
    public boolean contains(double testX, double testY) {
      return false;
      // TODO: left as an exercise
    }
  }
```

Creating and Interacting with `Circle` objects

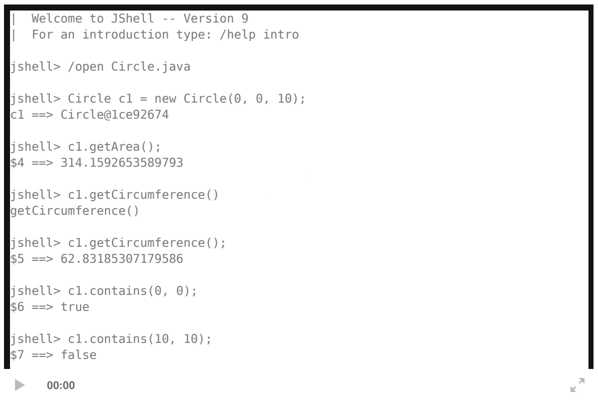
To use the `Circle` class, we can either:

- create a `main()` function, compile and link with the `Circle` class, and create an executable program, just like we usually do with a C program, OR
- use `jshell`, which is part of Java 9 (but not earlier versions). `jshell` provides a *read-evaluate-print loop* (REPL) to help us quickly try out various features of Java.

We will write a complete Java program with `main()` later in this class, but for now, we will use `jshell` to demonstrate the various language features of Java [7 [#fn:8]].

The demonstration below loads the `Circle` class written above (with the `contains` method completed) from a file named `Circle.java` [8 [#fn:9]], and creates two `Circle` objects, `c1` and `c2`. We use the `new` keyword to tell Java to create an object of type `Circle` here, passing in the center and the radius.

```
Circle c1 = new Circle(0, 0, 100);
```



```
|  Welcome to JShell -- Version 9
|  For an introduction type: /help intro

jshell> /open Circle.java

jshell> Circle c1 = new Circle(0, 0, 10);
c1 ==> Circle@1ce92674

jshell> c1.getArea();
$4 ==> 314.1592653589793

jshell> c1.getCircumference()
getCircumference()

jshell> c1.getCircumference();
$5 ==> 62.83185307179586

jshell> c1.contains(0, 0);
$6 ==> true

jshell> c1.contains(10, 10);
$7 ==> false
```

Recorded with asciinema

Reference Type vs. Primitive Type

The variable `c1` actually stores an abstraction over a *reference* to the Circle object, instead of the object itself. *All objects are stored as references in Java.*

The other variable type supported in Java is *primitive* type. A variable of primitive type stores the *value* instead of a reference to the value. Java supports eight *primitive* data types: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` and `char`. If you are familiar with C, these data types should not be foreign to you. One important difference is that a `char` variable stores a 16-bit Unicode character, not an 8-bit character like in C. Java uses the type `byte` for that. The other notable difference is that Java defines `true` and `false` as possible value to a `boolean`, unlike C which uses `0` for false and non-`0` for true.

You can read all about Java variables [https://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html] and primitive data types [https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html] in Oracle's Java Tutorial.

Type Safety

Some languages are stricter in terms of type "compatibility" than others. C compilers, however, are not very strict. If it detects something strange with the type you used, it will issue a warning, but still let your code compiles and runs.

Take:

```
1  #include <stdio.h>
2  int main()
3  {
4      printf("%d\n", "cs2030");
5  }
```

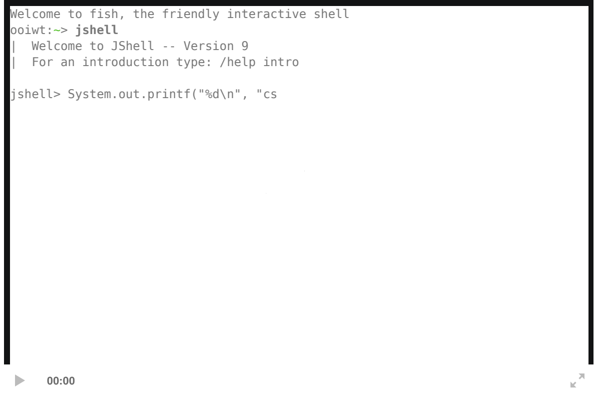
In Line 4, we treat the address to a string as integer. This generates a compiler's warning.

In C, you can *type cast* a variable from one type into another, i.e., force the compiler to treat a variable of one type as another type. The compiler would listen and do that for you. The following code would print out gibberish and would compile perfectly without error.

```
#include <stdio.h>
int main()
{
    printf("%d\n", (int)"cs2030");
}
```

Such flexibility and loose rules for type compatibility could be useful, if you know what you are doing, but for most programmers, it could be a major source of unintentional bugs, especially if one does not pay attention to compiler's warning or one forces the warning to go away without fully understanding what is going on.

Java is very strict when it comes to type checking, and is one of the *type-safe* languages. Java ensures that basic operations (such as `+`, `-`, etc) and method calls apply to values in a way that makes sense. If you try to pull the same trick as above, you will receive an error:



```
Welcome to fish, the friendly interactive shell
ooiwt:~> jshell
|  Welcome to JShell -- Version 9
|  For an introduction type: /help intro

jshell> System.out.printf("%d\n", "cs
```

00:00

Recorded with asciinema

Exercise

1. In the example above, we implemented a class `Circle`. There, we store and pass around two `double` variables that correspond to the x-coordinate and y-coordinate of a point. The code would be neater if we create a second class `Point` that encapsulates the concept of a point on a 2D plane and the operations on points.

   Implement a new class `Point` and modify the class `Circle` to use the class `Point`. Pay attention to what methods and fields (if any) you expose as `public` outside of the abstraction barrier of a `Point` object.

   You will need to use `jshell` from Java 1.9 (or JDK 9) to interact with your new classes.

2. Use `jshell` to try out the following.

```
class A {
    public static int x = 1;
    public int y = 5;

    void incrX() {
        x = x + 1;
    }

    void incrY() {
        y = y + 1;
    }
}

A a1 = new A();
A a2 = new A();
```

   After executing `a1.x = 10`, what is the value of `a2.x`?

   After executing `a1.y = 10`, what is the value of `a2.y`?

   Is `A.x = 3` a valid statement? Is `A.y = 3` a valid statement?

   Note: Even though `a1.x` is valid, it is considered a bad programming practice to access a class field through an instance variable (e.g., `a1.x`). The proper way to do it is to use the class name `A.x`).

3. Consider the following two classes:

```
1   class A {
2       private int x;
3     public void changeSelf() {
4       x = 1;
5     }
6     public void changeAnother(A a) {
7       a.x = 1;
8     }
9
10  }
11
12  class B {
13      public void changeAnother(A a) {
14          a.x = 1;
15      }
16  }
```

   Which line(s) above violate the `private` access modifier of `x`?

---

1. Often, the instructions and data are stored in different regions of the memory.

2. Javascript would happily convert `4` into a string for you, and return `45`.

3. assuming the parameters are passed correctly.

4. Computer scientists just can't decide on what to call this!

5. Others include `protected` and the *default* modifier. For beginners, it is better that we explicitly specify something as `private` or `public`.

6. The class `Math` is provided by the package `java.lang` in Java. A package is simply a set of related classes (and interfaces, but I have not told you what is an interface yet). To use this class, we need to add the line `import java.lang.Math` at the beginning of our program.

7. You can download and install `jshell` yourself, as part of Java Development Kit version 9 (JDK 9) [http://www.oracle.com/technetwork/java/javase/downloads/jdk9-downloads-3848520.html]

8. We use the convention of one public class per file, name the file with the exact name of the class (including capitalization), and include the extension `.java` to the filename.