Abstract Data Types, Encapsulation, and Object Oriented Programming

(Follows the Sebesta Text Chapters 11 and 12)

OO and ADTs

Most all modern programming languages support Object Oriented (OO) programming.

The hallmarks of OO are data abstraction and encapsulation.

- **Abstraction** is a way of looking at some entity that only focuses on the most significant parts.
- **Encapsulation** is a way of organizing code into units such that the units protect state and provide behavior.

OO and ADTs

We have learned about two types of abstraction in this class,

- **Process abstraction** occurs when we give a name to a sequence of statements. This allows us to call upon a sequence of statements instead of manually executing each statement ourselves. We see this with subprograms (functions). This was the first type of abstraction in Computer Science, and even Plankalkül supported it in the 1940s.
- **Data abstraction** is much newer, with the ideas for it beginning in the 1960s and COBOL. C also provided data abstraction with the concept of structs. This allows us to create composite types.

OO and ADTs

Abstract Data Types (ADTs) are enclosures that define a specific type of data *and* the subprograms that operate on that data type.

- Consider an ADT called `Spaceship`.
    - Includes instance variables (state) that every instance (every object) should have.
    - Defines methods that should be used to interact with an instance (behavior).

OO and ADTs

ADTs are defined in a single syntactic unit. This *does not* mean they are always defined in a single file though (think C++).

- ADTs have access controls that can hide information from outside the ADT instances.
- This is very important! It means that the user doesn't have to know anything about the underlying storage or function of the ADT.
- All the users need to know is how to use it.

OO and ADTs

Question: Are C structs objects? Are they ADTs?

```c
#include <stdio.h>
#include <stdlib.h>

struct spaceshp_functions;

typedef struct spaceship_t {
	int hp;
	double shield_level;
	char* name;
	struct spaceship_functions* f_table;
} Spaceship;

struct spaceship_functions {
	void (*str)(Spaceship* s);
	void (*set_hp)(Spaceship* s, int hp);
};

void set_hp(Spaceship* s, int hp){
	s->hp = hp;
}

void print_name(Spaceship* s){
	printf("%s - %d: %f\n", s->name, s->hp, s->shield_level);
}
```

```c
Spaceship* init(){
	return malloc(sizeof(Spaceship));
}

int main(int argc, char** argv){
	struct spaceship_functions funcs;
	funcs.str = &print_name;
	funcs.set_hp = &set_hp;
	Spaceship* firefly = init();
	firefly->f_table = &funcs;
	firefly->name = "Firefly";
	firefly->f_table->set_hp(firefly, 1000);
	firefly->shield_level = 1.0f;
	firefly->f_table->str(firefly);
	free(firefly);
}
```

OO and ADTs

- Through the rest of this semester we will use the term ADT to refer to the construct or definition of an ADT (the class). We will use the term **object** to refer to a particular instance of an ADT.
- We will use the term **clients** to refer to the program units that use a specific ADT.

ADTs

We use ADTs as a means of making programs **less complex**.

- It allows us to separate code into areas of concern.
- Instead of concentrating on the big picture of what the program is trying to accomplish, we focus on writing robust definitions of the smaller units of which the program is composed.
- Ideally, we then get high code reusability out of the smaller solutions.

ADTs

An important note: Even simple data types *can be* ADTs. We tend to forget this when we think of built-in types, but even built-in types are sometimes ADTs.

- Consider a floating point number in Python. We don't have to understand how it is stored; in fact there have been multiple different ways to store a floating point number that systems have used in the past. All we really care about is how to use one.
- In C though, we can manipulate the bits. No data protection so no ADT.
- Even the concept of the byte is an ADT; in reality it is just the turning on or off of some switch. We don't care how that happens - just that we can use one.

ADTs

Information hiding is the concept that we can use an ADT without knowing how it works internally.

This implies:

- that ADTs can be very robust. By hiding the information of implementation we prohibit users from "damaging" our internal data.
- that our code can still work the same regardless of the system on which we are running. For instance, if we run on a system that doesn't follow the same standard for the storage of floating point numbers, we can still use floating point numbers. How they are stored on the systems is not important to us.

ADTs

Other benefits of information hiding (besides increased integrity of the data) are

- the amount of variables and code the programmer has to be aware of is reduced.
- name conflicts are less likely (scope of the variables are smaller).
- changes in how the ADT was programmed do not cause us to have to rewrite all code that uses the ADT.

ADTs

We need to be careful about designing our ADTs.

- If we aren't careful about adhering to proper practice we run the risk of losing the benefits of information hiding.
- Therefore we should (in general) not set access to our data as public.
- Instead, we should always use accessors.
  - But, don't get carried away.
  - We are generally talking about using "other people's" code here. (Java Example next slide).

```java
public class Color {
        private int red;
        private int green;
        private int blue;

        public Color(int red, int green, int blue){
                this.red = setValue(red);
                this.green = setValue(green);
                this.blue = setValue(blue);
        }

        public Color(Color c){
                this.red = c.getRed();
                this.green = c.getGreen();
                this.blue = c.getBlue();
        }

        public int getRed(){
                return this.red;
        }

        public int getGreen(){
                return this.green;
        }

        public int getBlue(){
                return this.blue;
        }

        int setValue(int color){
                if(color < 0 || color > 255){
                        throw new IllegalArgumentException("Values must be 0-255.");
                }
                return color;
        }
}
```

```java
public class Color {
        private int red;
        private int green;
        private int blue;

        public Color(int red, int green, int blue){
                this.red = setValue(red);
                this.green = setValue(green);
                this.blue = setValue(blue);
        }

        public Color(Color c){
                // Better!
                this.red = c.red;
                this.green = c.green;
                this.blue = c.blue;
        }

        public int getRed(){
                return this.red;
        }

        public int getGreen(){
                return this.green;
        }

        public int getBlue(){
                return this.blue;
        }

        int setValue(int color){
                if(color < 0 || color > 255){
                        throw new IllegalArgumentException("Values must be 0-255.");
                }
                return color;
        }
}
```

ADTs

If you aren't the one who wrote the code, you should probably use accessors. The reasons accessors are better are

- we can have read-only data by having a getter but no setter.
- we can include constraints in our setters. For example we might want to limit possible changes to our state to certain value ranges.
- (as already mentioned) changing the way the ADT is implemented does not affect clients.

ADTs

For a language to be able to provide ADTs it must

- provide a syntactic unit to enclose the declaration of the type and prototypes of the subprograms that implement the operations.
- decide whether the ADT must be specified with its interface and implementation together (Java, C++) or separately (C++).
- a means to make subprograms (methods) visible to clients so they can ask the ADT to manipulate data on its behalf.
- must provide external visibility for the name, but hide the representation.
- define the types of access control provided and how those access controls are specified.

ADTs

Sometimes, we can create data protection where it isn't explicitly given. Consider JavaScript - it provides OO facilities (but using Object-Based Inheritance instead of Class-Based Inheritance which we will discuss in a moment).

- JavaScript has no direct ability to mark data as private.
- We can simulate it though, by defining variables local to a function, and then returning an inner function from the same scope

```
class MyClass {
  constructor(){
    let myData = 0;

    this.changeData = (newValue) => {
      myData = newValue;
    }

    this.getData = () => {
      return myData;
    }
  }
}
```

This is called a **closure**. It works because both the function *and* the function's environment are exported together.

ADTs

In addition to the syntactic unit issues, language designers must decide if an ADT can be parameterized.

- Parameterizing an ADT means specializing it so that can hold specific data types.
- We have seen parameterized types in Java; for instance when defining an ArrayList<>.
  - ArrayLists<> require us to denote the type of ArrayList we wish to use by filling in the "<>" section of the declaration.
- This is important for generic programming.

ADTs

In C++ for instance, we could make a generic stack (note C++ does have a std::stack<>. This is NOT that structure.):

```
template <typename T>
class Stack {

  public:
    void push(T const & e);
    void pop();
    T top() const;

  private:
    std::vector<T> elements;
};
```

ADTs

We would then be able to create an object of this type by parameterizing our declaration:

```
Stack<int> intStack;
Stack<float> floatStack;
Stack<Concert> concertStack;
```

Inheritance

We have two ways of making new data types in OO programming.

- **Composition**: New types are made by combining existing types.
- **Inheritance**: New types are made by customizing existing types.

You've probably heard someone complain about OOP. Often, they are referring to inheritance misuse.

Inheritance

Remember what we said at the beginning of these slides - OO was created to **reduce complexity**. Now consider the methods at

https://docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html

Inheritance

In general, we want to follow the **Liskov Substitution Principle**, which states

> An object should be able to be replaced by an object of one of its subtypes without breaking the program.

By way of analogy, "Imagine you have a car class. It has a brake, a throttle, and a steering wheel function. You create subtypes; one is front-wheel drive, and one is rear-wheel drive. Pressing the throttle on both makes the car accelerate. You make another subtype that is all-wheel drive. When you press the throttle on this car, it should also accelerate. If it locks the doors and blows-up, you've violated the principle." More classic example:

What's the problem with this?

```java
class Rectangle {
    protected int width;
    protected int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int getWidth() {
        return width;
    }

    public void setWidth(int width) {
        this.width = width;
    }

    public int getHeight() {
        return height;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int calculateArea() {
        return width * height;
    }
}
```

```java
class Square extends Rectangle {
    public Square(int side) {
        super(side, side);
    }

    @Override
    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    @Override
    public void setHeight(int height) {
        super.setWidth(height);
        super.setHeight(height);
    }
}
```

Inheritance

Languages that allow multiple inheritance can also suffer from the **Diamond Problem**.

Consider:

```python
class GameObject:
  def __init__(self, x, y):
    self.__x = x
    self.__y = y

class Updateable:
  def update(delta):
    pass

class Drawable:
  def draw():
    pass

class DUGameObject(Updateable, Drawable):
  def __init__(self):
    pass
```

Inheritance

There are two ways of implementing inheritance.

- **Classical** - Also called class-based inheritance. Classes are blueprints. We create subtypes by composition and overriding of parent symbols (Java, C++, Python).
- **Object** - Sometimes called prototype-based inheritance. No classes, only objects. We inherit by cloning or copying existing objects (JavaScript, Self, Lua).

Classical:

```
class Employee:
  def __init__(self, name):
    self.name = name

  def calc_pay(self):
    pass

class HourlyEmployee(Employee):
  def __init__(self, name):
    super().__init__(name)
    self.hours_worked = 0
    self.wage = 20.00

  def calc_pay(self):
    return self.wage * self.hours_worked

class SalaryEmployee(Employee):
  def __init__(self, name):
    super().__init__(name)
    self.salary = 60000.00

  def calc_pay(self):
    return self.salary / 52
```

Object Based:

```
function Employee(name){
        return {
                name: name,
                toStr: function(){
                        return name + " $" + this.calc_pay();
                }
        }
}

function HourlyEmployee(name){
        let tmp = Employee(name);
        tmp.wage = 20.0;
        tmp.hours_worked = 40;
        tmp.calc_pay = function(){
                return this.wage * this.hours_worked;
        }
        return tmp;
}

function SalaryEmployee(name){
        let tmp = Employee(name);
        tmp.salary = 60000;
        tmp.calc_pay = function(){
                return this.salary / 52;
        }
        return tmp;
}

edith = HourlyEmployee("Edith");
joanne = SalaryEmployee("Joanne");
console.log(edith.calc_pay());
console.log(joanne.calc_pay());
```

Inheritance

Object based is much more flexible and highly writable.

- Can be harder to read and therefore maintain.
- Often used in dynamic languages; problem of readability is exacerbated by lack of static type checking.
- Doesn't enforce structure or heirarchy the way class-based does.

Ultimately, we have to keep in mind that everything we are given is a tool. Misusing a tool is the programmer's fault, not the tool's fault.