


The Daily PL - 6/29/2023

We turn the page from imperative programming and begin our work on object-oriented programming!

The Definitions of Object-Oriented Programming

We can define object-oriented programming using four different definitions:

1. A language with support for abstraction of abstract data types (ADTs) (yes, I do mean to say that twice!). (from Sebesta)
2. A language with support for objects which are containers of data (attributes, properties, fields, etc.) and code (methods). (from [Wikipedia](https://en.wikipedia.org/wiki/Object-oriented_programming)  https://en.wikipedia.org/wiki/Object-oriented_programming.)
3. Pertaining to a technique or a programming language that supports objects, classes, and inheritance; Some authorities list the following requirements for object-oriented programming: information hiding or encapsulation, data abstraction, message passing, polymorphism, dynamic binding, and inheritance. (from ISO in standard 2382:2015)

As graduates of CS1021C and CS1080C, the second definition is probably not surprising. The first definition, however, leaves something to be desired. Using Definition (1) means that we have to a) know the definition of *abstraction* and *abstract data types* and b) know what it means to apply abstraction to ADTs.

Abstraction (Reprise)

There are two fundamental types of abstractions in programming: process and data. We have talked about the former but the latter is new. When we talked previously about process abstractions, we did attempt to define the term abstraction and even had some success. Let's review that discussion and our conclusions.

Sebesta formally defines *abstraction* as the view or representation of an entity that includes only the most significant attributes. This definition seems to align with our notion of abstraction especially the way we use the term in phrases like "abstract away the details." It didn't feel like a good definition to me until I thought of it this way:

Consider that you and I are both humans. As humans, we are both carbon based and have to breath to survive. But, we may not have the same color hair. I can say that I have red hair and you have blue hair to point out the significant attributes that distinguish us. I need not say that we are both carbon based and have to breath to survive because we are both human and we have "abstracted away" those facts into our common humanity.

There's another interesting definition of Abstraction that I think works slightly better. Let's try it on for size:

Abstraction: Removing the detail to simplify and focus attention on the essence.

Oh, I like that! Or, if that doesn't suit your fancy, how about this:

Abstraction: Remembering what is important in a given context and forgetting what's not.

That's pithy!

(Kramer, Jeff. Is Abstraction the Key to Computing? *Communications of the ACM*. April 2007. Vol. 50, No. 4)

Abstract Data Types (ADTs)

The second form of abstraction available to programmers is: *data abstraction*. As functions, procedures and methods are the syntactic and semantic means of abstracting processes in programming languages, ADTs are the syntactic and semantic means of abstracting *data* in programming languages. ADTs combine (*encapsulate*) data (usually called the ADT's *attributes*, *properties*, etc) and operations that operate on that data (usually called the ADT's *methods*) into a single entity.

Hiding is a significant advantage of ADTs. ADTs hide the data being represented and allow that its manipulation only through pre-defined methods, the ADT's *interface*. The interface typically gives the ADT's user the ability to manipulate/access the data internal to the type and perform other semantically meaningful operations (e.g., sorting a list).

Some common ADTs are:

1. Stack
2. Queue
3. List
4. Array
5. Dictionary
6. Graph
7. Tree
8. Heap

These are so-called *user-defined ADTs* because they are defined by the user (deep, huh?) of a programming language and composed of primitive data types.

Here's a thinker: Are primitives a type of ADT? A primitive type like floating point numbers would seem to meet the definition of an abstract data type:

1. It's underlying representation is hidden from the user (the programmer does not care whether FPs are represented according to IEEE754 or some other specification)
2. There are operations that manipulate the data (addition, subtraction, multiplication, division).

Parameterized ADTs

One of the most *fundamental* ADTs in the computer scientist's tool chest is the stack. I am sure that you have all worked with a stack before. The stack *contains* a list of items chronologically ordered according to the time that they were added to the list. The stack is a last-in-first-out (LIFO) data structure. The stack supports the

1. top: Retrieve a copy of the item most recently inserted into the list.
2. pop: Retrieve the item most recently inserted into the list and remove it from the list.
3. push: Add a new element to the list.

As developers we obviously want to know, just *what type* of elements can we store in this "list"? Well, anything really, we just need to write the code appropriately. Let's say that we are writing a simple integer calculator so we will need a stack of numbers. In Java, we would have something that looked like this:

```
package edu.uc.cs.cs3003;

import java.util.ArrayList;
import java.util.EmptyStackException;
import java.util.List;

public class Stack {
    Stack() {
        m_data = new ArrayList<Integer>();
    }

    public void push(Integer newelement) {
        m_data.add(newelement);
    }

    public Integer top() throws EmptyStackException {
        if (m_data.size() == 0) {
            throw new EmptyStackException();
        }
        return m_data.get(m_data.size() - 1);
    }

    public Integer pop() throws EmptyStackException {
        if (m_data.size() == 0) {
            throw new EmptyStackException();
        }
        Integer top = top();
        m_data.remove(m_data.size() - 1);
        return top;
    }

    private List<Integer> m_data;
}
```

Great. We have all the operations that we need. We can easily use that data structure in a Java program like this:

```
package edu.uc.cs.cs3003;

import java.util.EmptyStackException;

/**
 * Hello world!
 */
public final class App {

    private static void TestStack() {
        Stack stack = new Stack();

        stack.push(5);
        stack.push(10);
        stack.push(15);
        stack.push(20);

        System.out.println(stack.pop());
        System.out.println(stack.pop());
        System.out.println(stack.pop());
        System.out.println(stack.pop());

        try {
            System.out.println(stack.pop());
        } catch (EmptyStackException ese) {
            System.out.println("Oops: " + ese.toString());
        }
    }

    public static void main(String[] args) {
        TestStack();
    }
}
```

When we run it, we see the output:

20

15

10

5

Oops: java.util.EmptyStackException


Perfect! Exactly what we wanted. Even the logical error that we (I) committed is caught.

There's just one. small. problem. What do we do if there comes a time when I want to use my Stack ADT in an application that is not a calculator? In other words, I will need the Stack ADT to contain elements that are, say, characters when I am using a Stack ADT to determine if two strings are palindromic.

I guess I could just write the entire ADT again. I could name it something like CharacterStack. That just leads us to another question: What if, now, I need a Stack ADT that will let me contain

elements that are Pancakes (what better thing in the world is there than a stack of pancakes?). I rewrite the Stack ADT and name it something like PancakeStack ... where does the craziness stop??

It stops right here, with *parameterized ADTs*: An ADT that can store elements of any type.

(alternate definition: An ADT parameterized with respect to the component type [\[courtesy\]](http://www.cs.kent.edu/~durand/CS43101Fall2004/DT-UserDefinedADT.html)  (<http://www.cs.kent.edu/~durand/CS43101Fall2004/DT-UserDefinedADT.html>)).

In Java, these parameterized ADTs are known as *Generics* and they look like templates in C++. Here is how we would write the same Stack in a generic manner:

```
package edu.uc.cs.cs3003;

import java.util.ArrayList;
import java.util.EmptyStackException;
import java.util.List;

public class GStack<ElementType> {
    GStack() {
        m_data = new ArrayList<ElementType>();
    }

    public void push(ElementType newelement) {
        m_data.add(newelement);
    }

    public ElementType top() throws EmptyStackException {
        if (m_data.size() == 0) {
            throw new EmptyStackException();
        }
        return m_data.get(m_data.size() - 1);
    }

    public ElementType pop() throws EmptyStackException {
        if (m_data.size() == 0) {
            throw new EmptyStackException();
        }
        ElementType top = top();
        m_data.remove(m_data.size() - 1);
        return top;
    }

    private List<ElementType> m_data;
}
```

And if I wanted to use that to create my stack of characters for the palindrome-detection application, I would instantiate it this way:

```
private static void TestGStack() {
    GStack<Character> stack = new GStack<Character>();

    stack.push('a');
    stack.push('b');
    stack.push('c');
    stack.push('d');

    System.out.println(stack.pop());
}
```


```
System.out.println(stack.pop());
System.out.println(stack.pop());
System.out.println(stack.pop());

try {
    System.out.println(stack.pop());
} catch (EmptyStackException ese) {
    System.out.println("Oops: " + ese.toString());
}
}
```

And look at this amazing output:

```
d
c
b
a
Oops: java.util.EmptyStackException
```

Absolutely perfect!

If you want to see all the code for this demo so that you can explore it yourself, it's online [here](https://github.com/hawkinsw/cs3003/tree/main/adts/stack)  (<https://github.com/hawkinsw/cs3003/tree/main/adts/stack>).

Overriding in OOP

Recall the concept of inheritance that we discussed previously. Besides its utility as a formalism that describes the way *a language supports abstraction of ADTs* (and, therefore, makes it a nominally OO language), inheritance provides a practical benefit in software engineering. Namely, it allows developers to build hierarchies of types.

Hierarchies are composed of pairs of classes -- one is the superclass and the other is the subclass. A superclass could conceivably be itself a subclass (to some other class). A subclass could itself be a superclass (to some other class). In terms of a family tree, we could say that the subclass is a descendant of the superclass (Note: remember that the terms superclass and subclass are not always the ones used by the languages themselves; C++ refers to them as base and derived classes, respectively).

A subclass inherits both the data and methods from its superclass(es). However, as Sebesta says, sometimes "... the features and capabilities of the [superclass] are not quite right for the new use." *Overriding* methods allows the programmer to keep most of the functionality of the superclass and customize the parts that are "not quite right."

An *overridden* method is defined in a subclass and replaces the method with the same name (and usually protocol) in the parent.

The official documentation and tutorials for Java describe overriding in the language this way: **"An instance method in a subclass with the same signature (name, plus the number and the**

type of its parameters) and return type as an instance method in the superclass overrides the superclass's method." [⇒ \(https://docs.oracle.com/javase/tutorial/java/landl/override.html\)](https://docs.oracle.com/javase/tutorial/java/landl/override.html)

The exact rules for overriding methods in Java are **online at the language specification** [⇒ \(https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.4.8.1\)](https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.4.8.1).

Let's make it concrete with an example:

```
class Car {
    protected boolean electric = false;
    protected int wheels = 4;

    Car() {
    }

    boolean ignite() {
        System.out.println("Igniting a generic car's engine!");
        return true;
    }
}


class Tesla extends Car {
    Tesla() {
        super();
        electric = true;
    }

    @Override
    boolean ignite() {
        System.out.println("Igniting a Tesla's engine!");
        return true;
    }
}

class Chevrolet extends Car {
    Chevrolet() {
        super();
    }

    @Override
    boolean ignite() {
        System.out.println("Igniting a Chevrolet's engine!");
        return false;
    }
}
```

In this example, `Car` is the superclass of `Tesla` and `Chevrolet`. The `Car` class defines a method named `ignite`. That method will *ignite* the engine of the car -- an action whose mechanics differ based on the car's type. In other words, this is a perfect candidate for overriding. Both `Tesla` and `Chevrolet` implement a method with the same name, return value and parameters, thereby meeting Java's requirements for overriding. In Java, the `@Override` is known as an annotation. Annotations are **"a form of metadata [that] provide data about a program that is not part of the program itself."** [⇒ \(https://docs.oracle.com/javase/tutorial/java/annotations/\)](https://docs.oracle.com/javase/tutorial/java/annotations/) Annotations in Java are attached to particular syntactic units. In this case, the `@Override` annotation is attached to a method and it tells the compiler that the method is overriding a method from its superclass. If the compiler does not find a method in the superclass(es) that is capable of being overridden by the

method, an error is generated. This is a good check for the programmer. (Note: C++ offers similar functionality through the **override specifier**  <https://en.cppreference.com/w/cpp/language/override>.)

Let's say that the programmer actually implemented the **Tesla** class like this:

```
class Tesla extends Car {
    Tesla() {
        super();
        electric = true;
    }

    @Override
    boolean ignite(int testing) {
        super.ignite();
        System.out.println("Igniting a Tesla's engine!");
        return true;
    }
}
```

The **ignite** method implemented in **Tesla** *does not* override the **ignite** method from **Car** because it has a different set of parameters. The **@Override** annotation tells the compiler that the programmer thought they were overriding something. An error is generated and the programmer can make the appropriate fix. Without the **@Override** annotation, the code will compile but produce incorrect output when executed.

Assume that the following program exists:

```
public class CarDemo {
    public static void main(String args[]) {
        Car c = new Car();
        Car t = new Tesla();
        Car v = new Chevrolet();

        c.ignite();
        t.ignite();
        v.ignite();
    }
}
```

This code instantiates three different cars -- the first is a generic **Car**, the second is a **Tesla** and the third is a **Chevrolet**. Look carefully and note that the type of each of the three is actually stored in a variable whose type is **Car** and not a more-specific type (ie, **Tesla** or **Chevy**). This is not a problem because of dynamic dispatch (remember that?). At runtime, the JVM will find the proper **ignite** function and invoke it according to the variable's *actual* type and not its static type. Because **ignite** is overridden by **Chevy** and **Tesla**, the output of the program above is:

```
Igniting a generic car's engine!
Igniting a Tesla's engine!
Igniting a Chevrolet's engine!
```

Most OOP languages provide the programmer the option to invoke the method they are overriding

from the superclass. Java is no different. If an overriding method implementation wants to invoke the functionality of the method that it is overriding, it can do so using the `super` keyword.

```
class Tesla extends Car {
    Tesla() {
        super();
        electric = true;
    }

    @Override
    boolean ignite() {
        super.ignite();
        System.out.println("Igniting a Tesla's engine!");
        return true;
    }
}

class Chevrolet extends Car {
    Chevrolet() {
        super();
    }

    @Override
    boolean ignite() {
        super.ignite();
        System.out.println("Igniting a Chevrolet's engine!");
        return false;
    }
}
```

With these changes, the program now outputs:

```
Igniting a generic car's engine!
Igniting a generic car's engine!
Igniting a Tesla's engine!
Igniting a generic car's engine!
Igniting a Chevrolet's engine!
```

What if the programmer does not want a subclass to be able to customize the behavior of a certain method? For example, no matter how you subclass `Dog`, its noise method is always going to bark -- no inheriting class should change that. Java provides the `final` keyword to guarantee that the implementation of a method cannot be overridden by a subclass. Let's change the code for the classes from above to look like this:

```
class Car {
    protected boolean electric = false;
    protected int wheels = 4;

    Car() {
    }

    void start() {
        System.out.println("Starting a car ...");
        if (this.ignite()) {
            System.out.println("Ignited the engine!");
        } else {
            System.out.println("Did NOT ignite the engine!");
        }
    }
}
```

```
}

final boolean ignite() {
    System.out.println("Igniting a generic car's engine!");
    return true;
}

class Tesla extends Car {
    Tesla() {
        super();
        electric = true;
    }

    @Override
    boolean ignite() {
        super.ignite();
        System.out.println("Igniting a Tesla's engine!");
        return true;
    }
}

class Chevrolet extends Car {
    Chevrolet() {
        super();
    }

    @Override
    boolean ignite() {
        super.ignite();
        System.out.println("Igniting a Chevrolet's engine!");
        return false;
    }
}
```

Notice that `ignite` in the `Car` class has a `final` before the return type. This makes `ignite` a **final method** [↗\(https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.4.3.3\)](https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.4.3.3): "A method can be declared `final` to prevent subclasses from overriding or hiding it". (C++ has something similar -- **the `final` specifier** [↗\(https://en.cppreference.com/w/cpp/language/final\)](https://en.cppreference.com/w/cpp/language/final).) Attempting to compile the code above produces this output:


```
CarDemo.java:30: error: ignite() in Tesla cannot override ignite() in Car
    boolean ignite() {
        ^
    overridden method is final
CarDemo.java:43: error: ignite() in Chevrolet cannot override ignite() in Car
    boolean ignite() {
        ^
    overridden method is final
2 errors
```

Subclass vs Subtype


In OOP there is fascinating distinction between subclasses and subtypes. All those classes that inherit from other classes are considered subclasses. However, they are not all subtypes. For a type/class `S` to be a subtype of type/class `T`, the following must hold

Assume that $\phi(t)$ is some provable property that is true of t , an object of type T . Then $\phi(s)$ must be true as well for s , an object of type S .

This formal definition can be phrased simply in terms of behaviors: If it is possible to pass objects of type T as arguments to a function that expects objects of type S without any change in the behavior, then S is a subtype of T . In other words, a subtype behaves exactly like the "supertype".

Barbara Liskov who pioneered the definition and study of subtypes **put it this way**  (<https://doi.org/uc.idm.oclc.org/10.1145/62139.62141>): "If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T ."

Open Recursion

Open recursion in an OO PL is a fancy term for the combination of a) functionality that gives the programmer the ability to refer to the current object from within a method (usually through a variable named `this` or `self`) and b) dynamic dispatch. Thanks to open recursion, some method **A** of class **C** can call some method **B** of the same class. **But wait, there's more!**  (https://en.wikipedia.org/wiki/Ron_Popeil). Continuing our example, in open recursion, if method **B** is overridden in class **D** (a subclass of **C**), then the overridden version of the method is invoked when called from method **A** on an object of type **D** *even though method A is only implemented by class C*. Wild! It is far easier to see this work in real life than talk about it abstractly. So, consider our cars again:

```
class Car {
    protected boolean electric = false;
    protected int wheels = 4;

    Car() {
    }

    void start() {
        System.out.println("Starting a car ...");
        if (this.ignite()) {
            System.out.println("Ignited the engine!");
        } else {
            System.out.println("Did NOT ignite the engine!");
        }
    }

    boolean ignite() {
        System.out.println("Igniting a generic car's engine!");
        return true;
    }
}

class Tesla extends Car {
    Tesla() {
        super();
        electric = true;
    }
}
```

```
@Override
boolean ignite() {
    System.out.println("Igniting a Tesla's engine!");
    return true;
}

class Chevrolet extends Car {
    Chevrolet() {
        super();
    }

    @Override
    boolean ignite() {
        System.out.println("Igniting a Chevrolet's engine!");
        return false;
    }
}
```

The `start` method is only implemented in the `Car` class. At the time that it is compiled, the `Car` class has no awareness of any subclasses (ie, `Tesla` and `Chevrolet`). Let's run this code and see what happens:

```
public class CarDemo {
    public static void main(String args[]) {
        Car c = new Car();
        Car t = new Tesla();
        Car v = new Chevrolet();

        c.start();
        t.start();
        v.start();
    }
}
```

Here's the output:

```
Starting a car ...
Igniting a generic car's engine!
Ignited the engine!
Starting a car ...
Igniting a Tesla's engine!
Ignited the engine!
Starting a car ...
Igniting a Chevrolet's engine!
Did NOT ignite the engine!
```

Wow! Even though the implementation of `start` is entirely within the `Car` class and the `Car` class knows nothing about the `Tesla` or `Chevrolet` subclasses, when the `start` method is invoked on object's of those types, the call to `this`'s `ignite` method triggers the execution of code specific to the type of car!

How cool is that?

The Daily PL - 7/06/2023

Java Constructors in Subclasses

The Java Virtual Machine (the JVM) will insert an implicit call to the to-be-instantiated subclass' superclass' *default* constructor (i.e., the one with no parameters) if the to-be-constructed (sub)class does not do so explicitly. We'll make this clear with an example:

```
class Parent {
    Parent() {
        System.out.println("I am in the Parent constructor.");
    }

    Parent(int parameter) {
        System.out.println("This version of the constructor is not called.");
    }
}

class Child extends Parent {
    Child() {
        /*
         * No explicit call to super -- one is automatically
         * injected that invokes the parent constructor with no parameters.
         */
        System.out.println("I am in the Child constructor.");
    }
}

public class DefaultConstructor {
    public static void main(String args[]) {
        Child c = new Child();
    }
}
```

When this program is executed, it will print

```
I am in the Parent constructor.
I am in the Child constructor.
```

The main function is instantiating an object of the type `Child`. We can visually inspect that there is no explicit call to the `super()` from within the `Child` class' constructor. Therefore, the JVM will insert an implicit call to `super()` which actually invokes `Parent()`.

However, if we make the following change:

```
class Parent {
    Parent() {
        System.out.println("I am in the Parent constructor.");
    }

    Parent(int parameter) {
        System.out.println("This version of the constructor is not called.");
    }
}
```

```
}

class Child extends Parent {
    Child() {
        super(1);
        System.out.println("I am in the Child constructor.");
    }
}

public class DefaultConstructor {
    public static void main(String args[]) {
        Child c = new Child();
    }
}
```

Something different happens. We see that there is a call to `Child`'s superclass' constructor (the one that takes a single `int`-typed parameter). That means that the JVM will not insert an implicit call to `super()` and we will get the following output:

```
This version of the constructor is not called.
I am in the Child constructor.
```

A Different Way to OOP

So far we have talked about OOP in the context of Java. Java, and languages like it, are called Class-based OOP languages. In a Class-based OOP, classes and objects exist in different worlds. Classes are used to define/declare

1. the attributes and methods of an encapsulation, and
2. the relationships between them.

From these classes, objects are *instantiated* that contain those attributes and methods and respect the defined/declared hierarchy.

```
class Parent {
    Parent() {
        System.out.println("I am in the Parent constructor.");
    }

    Parent(int parameter) {
        System.out.println("This version of the constructor is not called.");
    }
}

class Child extends Parent {
    Child() {
        System.out.println("I am in the Child constructor.");
    }
}
```

We can see this in the example given above: The classes `Parent` and `Child` define (no) attributes and (no) methods and define the relationship between them. In `main()`, a `Child` is instantiated and stored in the variable `c`. `c` is an object of type `Child` that contains all the data associated

with a `Child` and a `Parent` and can perform all the actions of a `Child` and a `Parent`.

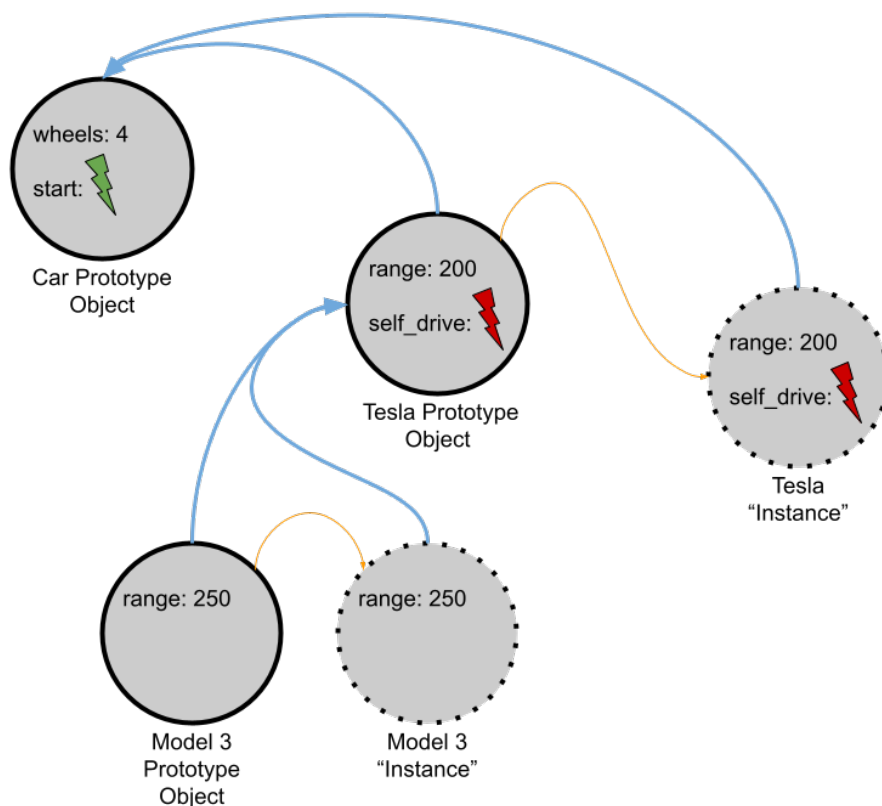
Nothing about Class-based OOP should be different than what you've learned in the past as you've worked with C++. There are several problems with Class-based OOP.

1. The supported attributes and methods of each class must be determined before the application is developed (once the code is compiled and the system is running, an object *cannot* add, remove or modify its own methods or attributes);
2. The inheritance hierarchy between classes must be determined before the application is developed (once the code is compiled, changing the relationship between classes will require that the application be recompiled!).

In other words, Class-based OOP does not allow the structure of the Classes (nor their relationships) to easily evolve with the implementation of a system.

There is another way, though. It's called Prototypal OOP. The most commonly known languages that use Prototypal OOP are JavaScript and Ruby! In Prototypal (which is a very hard word to spell!) OOP there is no distinction between Class and object -- everything is an object! In a Prototypal OOP there is a base object that has no methods or data attributes that every object is able to modify itself (its attributes and methods). To build a new object, the programmer simply copies from an existing object, the new object's so-called prototype, and customizes the copied object appropriately.

For example, assume that there is an object called `Car` that has one attribute (the number of wheels) and one method (start). That object can serve as the *prototype* car. To "instantiate" a new `Car`, the programmer simply copies the existing prototypical car object `Car` and gives it a name, say, `c`. The programmer can change the value of `c`'s number of wheels and invoke its method, start. Let's say that the same programmer wants to create something akin to a subclass of `Car`. The programmer would create a new, completely fresh object (one that has no methods or attributes), name it, say, `Tesla`, and link the new prototype `Tesla` object to the existing prototype car `Car` object through the prototype `Tesla` object's *prototype link* (the sequence of links that connects prototype objects to one another is called a *prototype chain*). If a Tesla has attributes (range, etc) or methods (self_drive) that the prototype car does not, then the programmer would install those methods on the prototype Tesla `Tesla`. Finally, the programmer would "declare" that the `Tesla` object is a prototype Tesla.



The blue arrows in the diagram above are prototype links. The orange lines indicate where a copy is made.

How does inheritance work in such a model? Well, it's actually pretty straightforward: When a method is invoked or an attribute is read/assigned, the runtime will search the prototype chain for the first prototypical object that has such a method or attribute. Mic drop. In the diagram above, let's follow how this would play out when the programmer calls `start()` on the Model 3 Instance. The Model 3 Instance does not contain a method named `start`. So, up we go! The Tesla Prototype Object does not contain that method either. All the way up! The Car Prototype Object, does, however, so that method is executed!

What would it look like to override a function? Again, relatively straightforward. If a Tesla performs different behavior than a normal Car when it starts, the programmer creating the Tesla Prototype Object would just add a method to that object with the name `start`. Then, when the prototype chain is traversed by the runtime looking for the method, it will stop at the `start` method defined in the Tesla Prototype Object instead of continuing on to the `start` method in the Car Prototype Object. (The same is true of attributes!)

There is (at least) one really powerful feature of this model. Keep in mind that the prototype objects are real things that can be manipulated at runtime (unlike classes which do not really exist after compilation) and prototype objects are linked together to achieve a type of inheritance. With reference to the diagram above, say the programmer changes the definition of the `start` method

on the Car Prototype Object. With only that change, any object whose prototype chain includes the Car Prototype Object will immediately have that new functionality (where it is not otherwise overridden, obviously) -- all without stopping the system!! How cool is that?

How scary is that? Can you imagine working on a system where certain methods you "inherit" change at runtime?



OOP or Interfaces?

Newer languages (e.g., Go, Rust, (new versions of) Java) are experimenting with new features that support one of the "killer apps" of OOP: The ability to define a function that takes a parameter of type *A* but that works just the same as long as it is called with an argument whose type is a *subtype* of *A*. The function doesn't have to care whether it is called with an argument whose type is *A* (or some subtype thereof) because the language's OOP semantics guarantee that anything the programmer can do with an object of type *A*, the programmer can do with an object of subtype of *A*.

Unfortunately, using OOP to accomplish such a feat may be like killing a fly with a bazooka.

Instead, modern languages are using a slimmer mechanism known as an interface or a trait. An interface just defines a list of methods that an implementer of that interface must support. Let's see some real Go code that does this -- it'll clear things up:

```
type Readable interface {  
    Read()  
}
```

This snippet defines an *interface* with one function (*Read*) that takes no parameters and returns no value. That interface is named `Readable`. Simple.

```
type Book struct {  
    title string  
}
```

This snippet defines a data structure called a `Book` -- such `struct`s are the closest that Go has to

classes.

```
func (book Book) Read() {  
    fmt.Printf("Reading the book %v\n", book.title)  
}
```

This snippet simply says that if variable `b` is of type `Book` then the programmer can call `b.Read()`. Now, for the payoff:

```
func WhatAreYouReading(r Readable) {  
    r.Read()  
}
```

This function only accepts arguments that *implement* (i.e., meet the criteria specified in the definition of) the `Readable` interface. In other words, with this definition, the code in the body of the function can safely assume that it can call `Read` on `r`. And, for the encore:

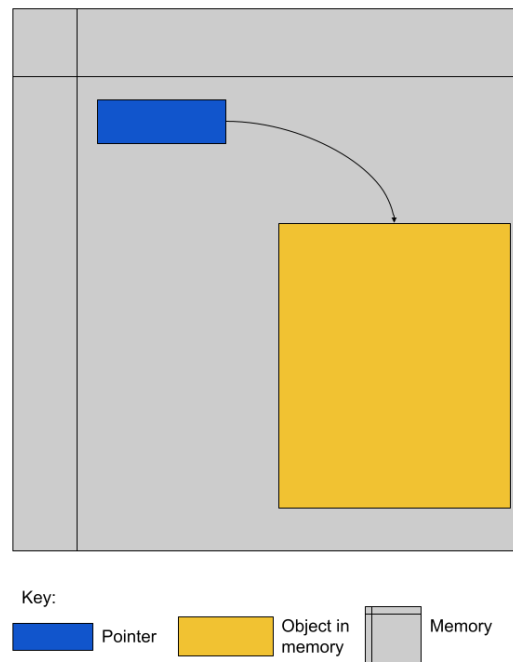
```
book := Book{title: "Infinite Jest"}  
WhatAreYouReading(book)
```


This code works exactly like you'd expect. `book` is a valid argument to `WhatAreYouReading` because it implements the `Read` method which, implicitly, means that it implements the `Readable` interface. But, what's really cool is that the programmer never had to say explicitly that `Book` implements the `Readable` interface! The compiler checks automatically. This gives the programmer the ability to generate a list of only the methods absolutely necessary for its parameters to implement to achieve the necessary ends -- and nothing unnecessary. Further, it decouples the person implementing a function from the person using the function -- those two parties do not have to coordinate requirements beforehand. Finally, this functionality means that a structure can implement as few or as many interfaces as its designer wants.

The Daily PL - 7/11/2023

Pointers

It is important to remember that pointers are like any other type -- they have a range of valid values and a set of valid operations that you can perform on those values. What are the range of valid values for a pointer? All valid memory addresses (but see below). And what are the valid operations? Addition, subtraction, dereference, and assignment (Note: the specific operations that are available for pointers may differ between languages).



In the diagram, the gray area is the memory of the computer. The blue box is a pointer. It points to the gold area of memory. It is important to remember that pointers and their targets *both* exist in memory! In fact, in true [Inception](https://www.imdb.com/title/tt1375666/) , a pointer can point to a pointer!

At the same time that pointers are types, they also have types. The type of a pointer includes the type of the target object. In other words, if the memory in the gold box held an object of type T, the blue box's type would be "pointer to type T." If the programmer *dereferences* the blue pointer, they will get access to the object in memory in the gold.


In an ideal scenario, it would always be the case that the type of the pointer and the type of the object at the target of the pointer are the same. However, that's not always the case.

When we were discussing the nature of the *type* of pointers, we specified that the range of valid values for a pointer are all memory addresses. In some languages this may be true. However,

some other languages specify that the range of valid values for a pointer are all memory addresses *and* a special *null* value that explicitly specifies a pointer does not point to a target.

We also discussed the operations that you can perform on a pointer-type variable. What we omitted from our list of valid operations (above) was a discussion of an operation that will fetch the address of a variable in memory. For languages that use pointers to support indirect addressing (see below), such an operation is required. In C/C++, this operation is performed using the *address of* (&) operator.

The Pros of Pointers

Though a very **famous and influential computer scientist**  (https://en.wikipedia.org/wiki/Tony_Hoare), once called his invention of *null references* a "billion dollar mistake" (he low balled it, I think!), the presence and power of pointers in a language is important for at least two reasons:

1. Without pointers, the programmer could not utilize the power of indirection.
2. Pointers give the programmer the power to address and manage heap-dynamic memory.

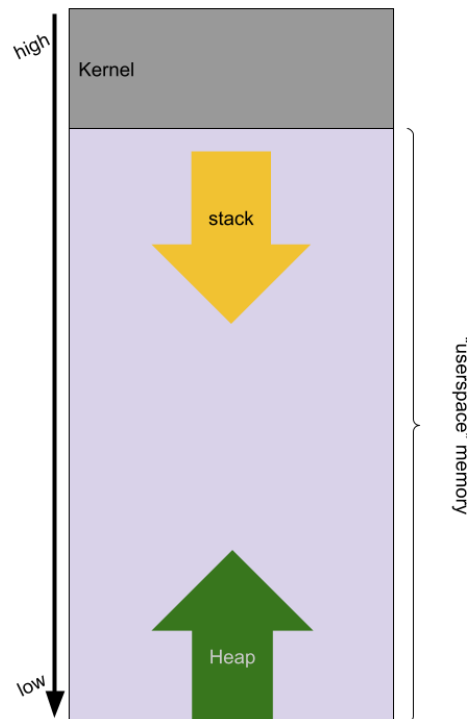
Indirection gives the programmer the power to link between different objects in memory -- something that makes writing certain data structures (like trees, graphs, linked lists, etc) easier. Management of heap-dynamic memory gives the programmer the ability to allocate, manipulate and deallocate memory at runtime. Without this power, the programmer would have to know *before execution* the amount of memory their program will require.

Heap-Dynamic Memory: A Digression

We have talked on more than one occasion about the *stack* and its utility in providing space for a local variable during the lifetime of that local variable. We have talked less (but have mentioned it, at least!) about the memory used to store static variables during the lifetime of those variables. And, we have not talked at all about the memory used to store heap-dynamic variables during the lifetime of those variables.

Given the term, it seems obvious that the storage for those variables will be in the heap. But just what is the heap? In a modern multiprocessing operating system, each program believes that they have complete access to the *entire* memory of the computer. In a sense they are right and in a sense they are wrong!

The operating system manages that illusion for them to make their lives easier. So, we'll go with the illusion, too. We learned earlier that that stack (the memory area and not the data structure) is both the place with storage for stack-local variables and the stack frames themselves and it grows down. Even though our programs don't have to coexist (knowingly) with other user applications, every process must make space for the kernel. Therefore, the kernel must own some space in memory so that it can continue to always be executing. That explains most of the figure below.



The only remaining item to locate is the Heap. The Heap is memory (and an area of memory) that exists on the opposite end of the address space from the stack. It grows in the opposite direction (as the operating system allocates pieces of it to programs/programmers for their use). Memory that is *dynamically allocated* while the program is executing comes from the heap (almost exclusively!). It is memory in this area that we will use pointers to address and keep in our grasp.

The Cons of Pointers

Their use as a means of indirection and managing heap-dynamic memory are powerful, but misusing either can cause serious problems.

Possible Problems when Using Pointers for Indirection

As we said in the last lecture, as long as a pointer targets memory that contains the expected type of object, everything is a-okay. Problems arise, however, when the target of the pointer is an area in memory that does not contain an object of the expected type (including garbage) and/or the pointer targets an area of memory that is inaccessible to the program.

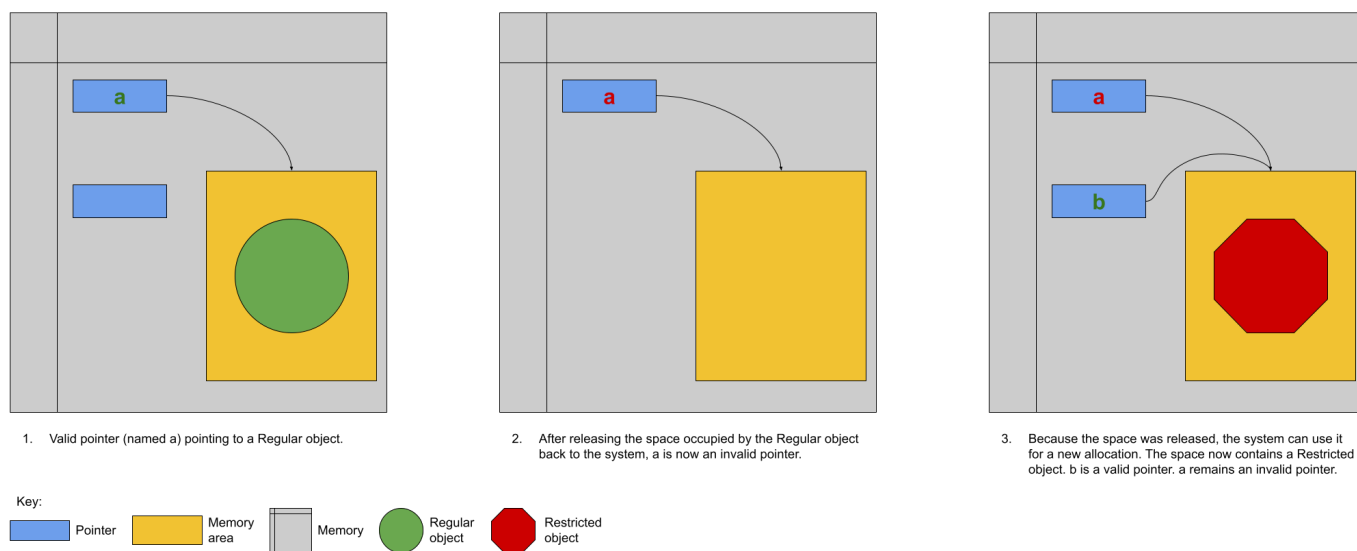
The former problem can arise when code in a program writes to areas of memory beyond their control (this behavior is usually an error, but is very common). It can also arise because of a *use after free*. As the name implies, a use-after-free error occurs when a programmer uses memory after it has been freed. There are two common scenarios that give rise to a use-after-free error:

1. Scenario 1:

1. One part of a program (part A) frees an area of memory that held a variable of type T that it no longer needs

2. Another part of the program (part B) has a pointer to that very memory
 3. A third part of the program (part C) overwrites that "freed" area of memory with a variable of type S
 4. Part B accesses the memory assuming that it still holds a variable of Type T
2. Scenario 2:
1. One part of a program (part A) frees an area of memory that held a variable of type T that it no longer needs
 2. Part A never nullifies the pointer it used to point to that area of memory though the pointer is now invalid because the program has released the space
 3. A second part of the program (part C) overwrites that "freed" area of memory with a variable of type S
 4. Part A incorrectly accesses the memory using the invalid pointer assuming that it still holds a variable of Type T

Scenario 2 is depicted visually in the following scenario and intimates why use-after-free errors are considered security vulnerabilities:



In the example shown visually above, the program's use of the invalid pointer means that the user of the invalid pointer can now access an object that is at a higher privilege level (Restricted vs Regular) than the programmer intended. When the programmer calls a function through the invalid pointer they expect that a method on the Regular object will be called. Unfortunately, a method on the Restricted object will be called instead. Trouble!

The latter problem occurs when a pointer targets memory beyond the program's control. This most often occurs when the program sets a variable's address to the special value that indicates the *absence* of value (e.g., `NULL`, `null`, `nil`) to indicate that it is invalid but later uses that pointer without checking its validity. For compiled languages this often results in the dreaded *segmentation fault* and for interpreted languages it often results in other anomalous behavior (like

Java's *Null Pointer Exception (NPE)*). Neither are good!

Possible Solutions

Wouldn't it be nice if we had a way to make sure that a pointer being dereferenced is valid so we don't fall victim to some of the aforementioned problems? What would be the requirements of such a solution?

1. Pointers to areas of memory that have been deallocated cannot be dereferenced.
2. The type of the object at the target of a pointer always matches the programmer's expectation.

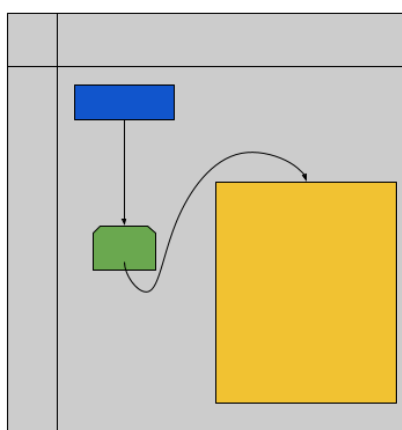
Your author describes two potential ways of doing this. First, are tombstones. Tombstones are essentially an intermediary between a pointer and its target. When the programming language implements pointers and uses tombstones for protection, a new tombstone is allocated for each pointer the programmer generates. The programmer's pointer targets the tombstone and the tombstone targets the pointer's actual target. The tombstone also contains an extra bit of information: whether it is valid. When the programmer first instantiates a pointer to some target *a* the compiler/interpreter

1. generates a tombstone whose target is *a*
2. sets the valid bit of the tombstone to *valid*
3. points the programmer's pointer to the tombstone.

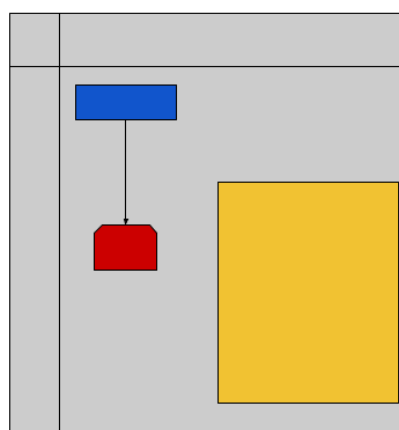
When the programmer dereferences their pointer, the compiler/runtime will check to make sure that the target tombstone's valid flag is set before doing the actual dereference of the ultimate target. When the programmer "destroys" the pointer (by releasing the memory at its target or by some other means), the compiler/runtime will set the target tombstone's valid flag to invalid. As a result, if the programmer later attempts to dereference the pointer after it was destroyed, the compiler/runtime will see that the tombstone's valid flag is invalid and generate an appropriate error.

This process is depicted visually in the following diagram.

Active Pointer (through Tombstone)



Inactive Pointer (through Tombstone)



Key:



Tombstone



Object in memory



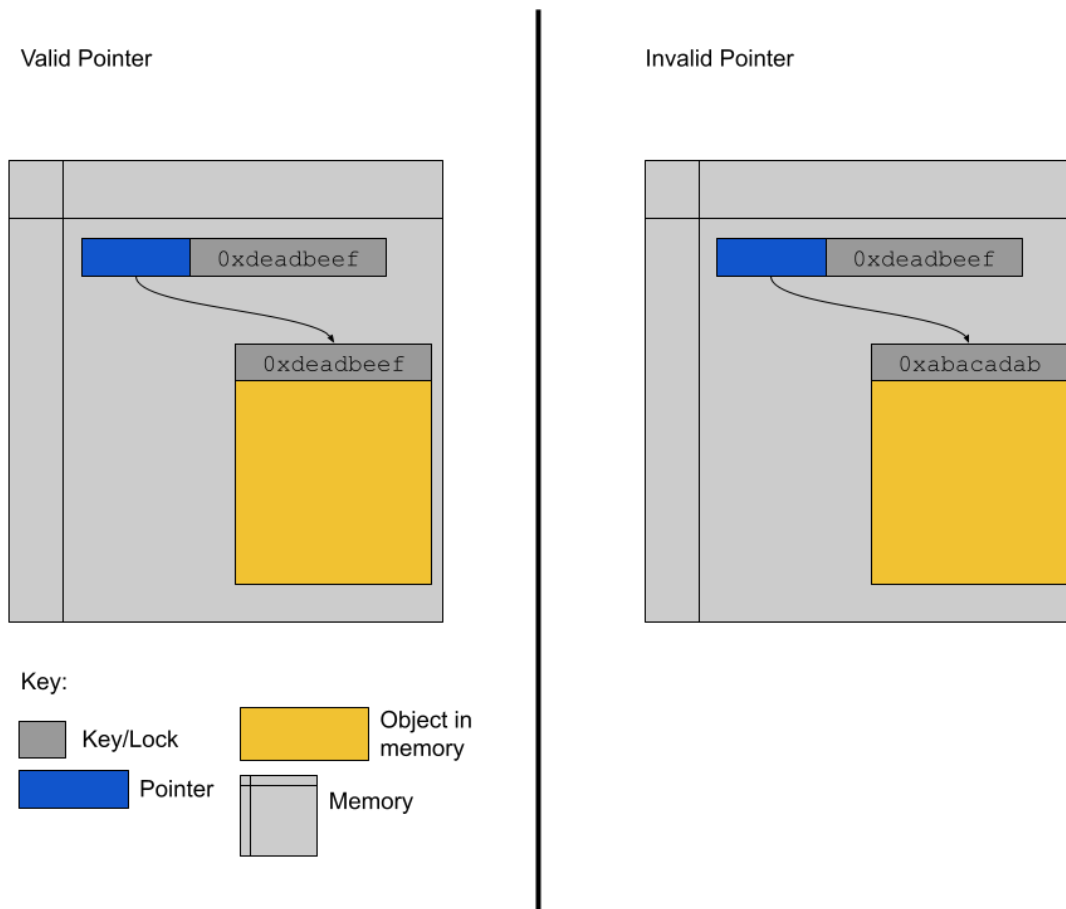
Pointer




Memory

This seems like a great solution! Unfortunately, there are downsides. In order for the tombstone to provide protection for the duration of the program's execution, once a tombstone has been allocated to protect a particular pointer it cannot be reclaimed. It must remain in place forever because it is always possible that a programmer can incorrectly reuse an invalid pointer *sometime real soon now*. As soon as the tombstone is deallocated, the protection that it provides is gone. The other problem is that the use of tombstones adds an additional layer of indirection to dereference a pointer and every indirection causes memory accesses. Though memory access times are small, they are not zero -- the cost of these additional memory accesses add up.

What about a solution that does not require an additional level of indirection? There is a so-called lock-and-key technique. This protection method requires that the pointer hold an additional piece of information beyond the address of the target: the key. The memory at the target of the pointer is also required to hold a key. When the system allocates memory it sets the keys of the pointer and the target to be the same value. When the programmer dereferences a pointer, the two keys are compared and the operation is only allowed to continue if the keys are the same. The process is depicted visually below.



With this technique, there is no additional memory access -- that's good! However, there are still downsides. First, there is a speed cost. For every dereference there must be a check of the equality of the keys. Depending on the length of the key that can take a significant amount of time. Second, there is a space cost. Every pointer and block of allocated memory now must have enough space to store the key. For systems where memory allocations are done in big chunks, the relative size overhead of storing, say, an 8 byte key is not significant. However, if the system allocates many small areas of memory, the relative size overhead is tremendous. Moreover, the more heavily the system relies on pointers the more space will be used to store keys rather than meaningful data.

Well, let's just make the keys smaller? Great idea. There's only one problem: The smaller the keys the fewer unique key values. Fewer unique key values mean that it is more likely an invalid pointer randomly points to a chunk of memory with a matching key (**pigeonhole principle**  (https://en.wikipedia.org/wiki/Pigeonhole_principle)).

Pointers for Dynamic Memory Management

As tools for dynamic memory management, the programmer can use pointers to target blocks (N.B.: I am using blocks as a generic term for memory and *am not* using it in the sense of a block [a.k.a. page] as defined in the context of operating systems) of dynamic memory that are allocated and deallocated by the operating system for use by an application. The programmer can use

these pointers to manipulate what is stored in those blocks and, ultimately, release them back to the operating system when they are no longer needed.

Memory in the system is a finite resource. If a program repeatedly asks for memory from the system without releasing previous allocations back to the system, there will come a time when the memory is exhausted. In order to be able to release existing allocations back to the operating system for reuse by other applications, the programmer must not lose track of those existing allocations. When there is a memory allocation from the operating system to the application that can no longer be reached by a pointer in the application, that memory allocation is *leaked*. Because the application no longer has a pointer to it, there is no way for the application to release it back to the system. Leaked memory belongs to the leaking application until it terminates.

For some programs this is fine. Some applications run for a short, defined period of time. However, there are other programs (especially servers) that are written specifically to operate for extended periods of time. If such applications leak memory, they run the risk of exhausting the system's memory resources and **failing** ➡ <https://shopify.engineering/17488512-most-memory-leaks-are-good>).

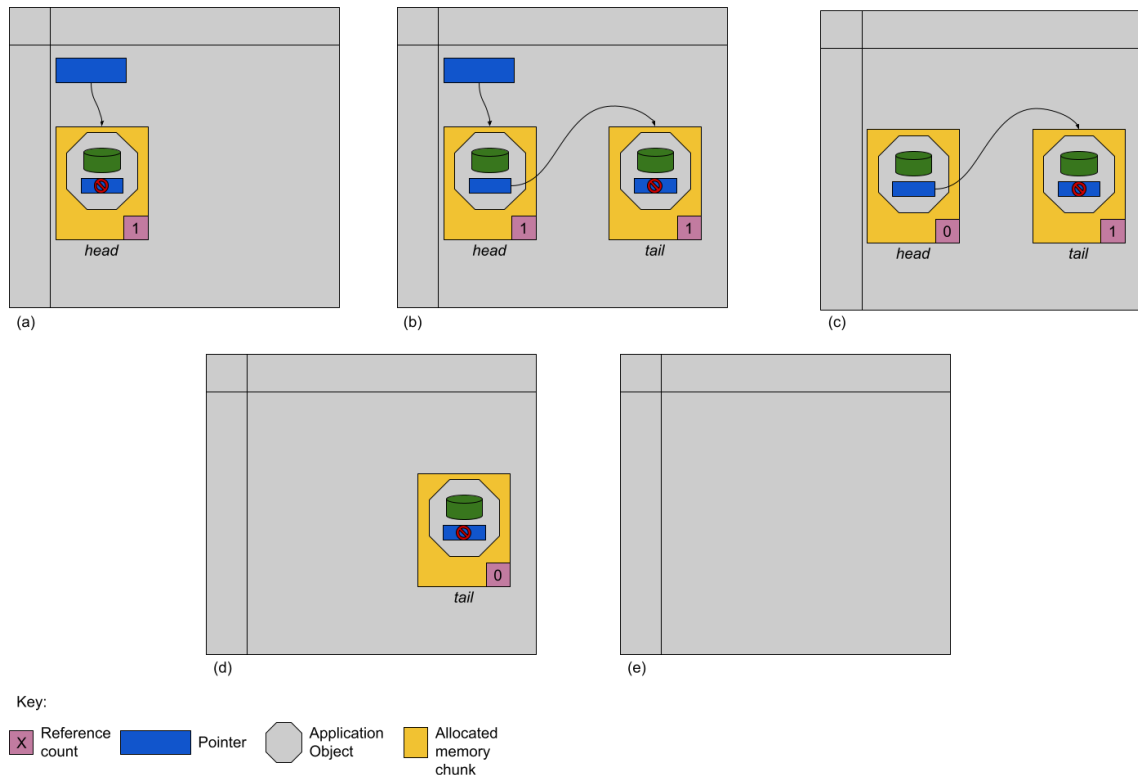
Preventing Memory Leaks

System behavior will be constrained when those systems are written in languages that do not support using pointers for dynamic memory management. However, what we learned (above) is that it is not always easy to use pointers for dynamic memory management correctly. What are some of the tools that programming languages provide to help the programmer manage pointers in their role as managers of dynamic memory.

Reference Counting

In a reference-counted memory management system, each allocated block of memory given to the application by the system contains a reference count. That *reference count*, well, counts the number of references to the object. In other words, for every pointer to an operating-system allocated block of memory, the reference count on that block increases. Every time that a pointer's target is changed, the programming language updates the reference counts of the old target (decrement) and the new target (increment), if there is a new target (the pointer could be changed to null, in which case there is no new target). When a block's reference count reaches zero, the language knows that the block is no longer needed, and automatically returns it to the system! Pretty cool.

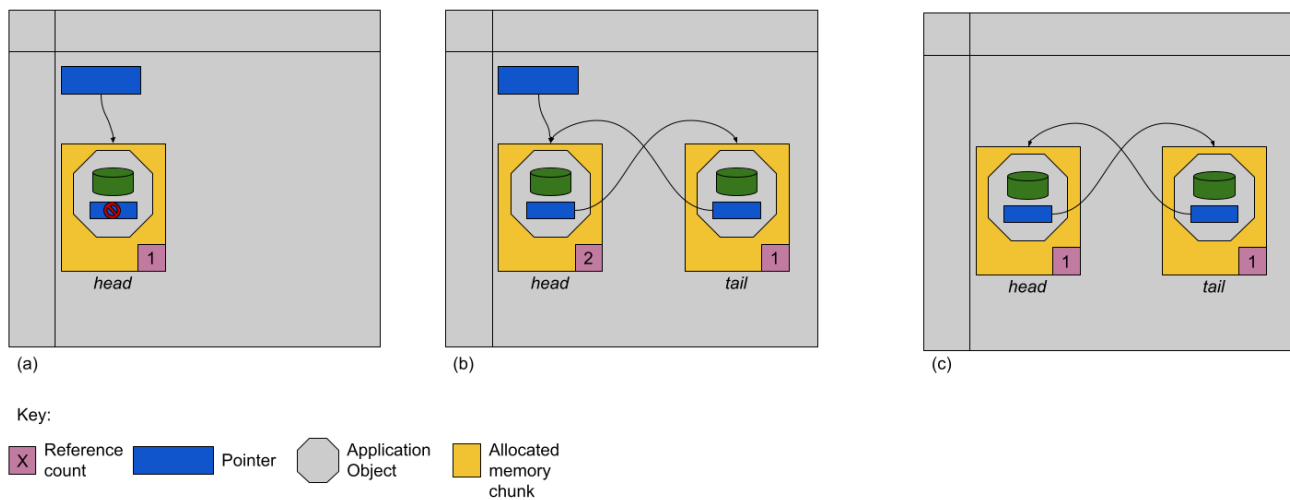
Memory Management via Reference Counting



The scenario depicted visually shows the reference counting process. At time (a), the programmer allocates a block of memory dynamically from the operating system and puts an application object in that block. Assume that the application object is a node in a linked list. The first node is the head of the list. Because the programmer has a pointer that targets that allocation, the block's reference count at time (a) is 1. At time (b), the programmer allocates a second block of memory dynamically from the system and puts a second application object in that block -- another node in the linked list (the tail of the list). Because the head of the list is referencing the tail of the list, the reference count of the tail is 1. At time (c) the programmer deletes their pointer (or reassigns it to a different target) to the head of the linked list. The programming language decrements the reference count of the block of memory holding the head node and deallocates it because the reference count has dropped to 0. Transitively, the pointer from the head application object to the tail application object is deleted and the programming language decrements the reference count of its target, the block of memory holding the tail application object (time (d)). The reference count of the block of memory holding the tail application object is now 0 and so the programming language automatically deallocates the associated storage (time (e)). Voila -- an automatic way to handle dynamic memory management.

There's only one problem. What if the programmer wants to implement a circularly linked list?

Memory Management via Reference Counting -- Cycles



Because the tail node points to the head node, and the head node points to the tail node, even after the programmer's pointer to the head node is deleted or retargeted, the reference counts of the two nodes will never drop to 0. In other words, even with reference-counted automatic memory management, there could still be a memory leak! Although there are algorithms to break these cycles, it's important to remember that reference counting is not a panacea. Python is a language that manages memory using reference counting.

Garbage Collection

Garbage collection (GC) is another method of automatically managing dynamically allocated memory. In a GC'd system, when a programmer allocates memory to store an object and no space is available, the programming language will stop the execution of the program (a so-called *GC pause*) to calculate the previously allocated memory blocks that are no longer in use and can be returned to the system. Having freed up space as a result of cleaning up unused garbage, the allocation requested by the programmer can be satisfied and the execution of the program can continue.

The most efficient way to engineer a GC'd system is if the programming language allocates memory to the programmer in fixed-size cells. In this scenario, every allocation request from the programmer is satisfied by a block of memory from one of several banks of fixed-size blocks that are stacked back-to-back. For example, a programming language may manage three different banks -- one that holds reserves of X-sized blocks, one that holds reserves of Y-sized blocks and one that holds reserves of Z-sized blocks. When the programmer asks for memory to hold an object that is of size a, the programming language will deliver a block that is just big enough to that object. Because the size of the requested allocation may not be exactly the same size as one of the available fixed-size blocks, space may be wasted.

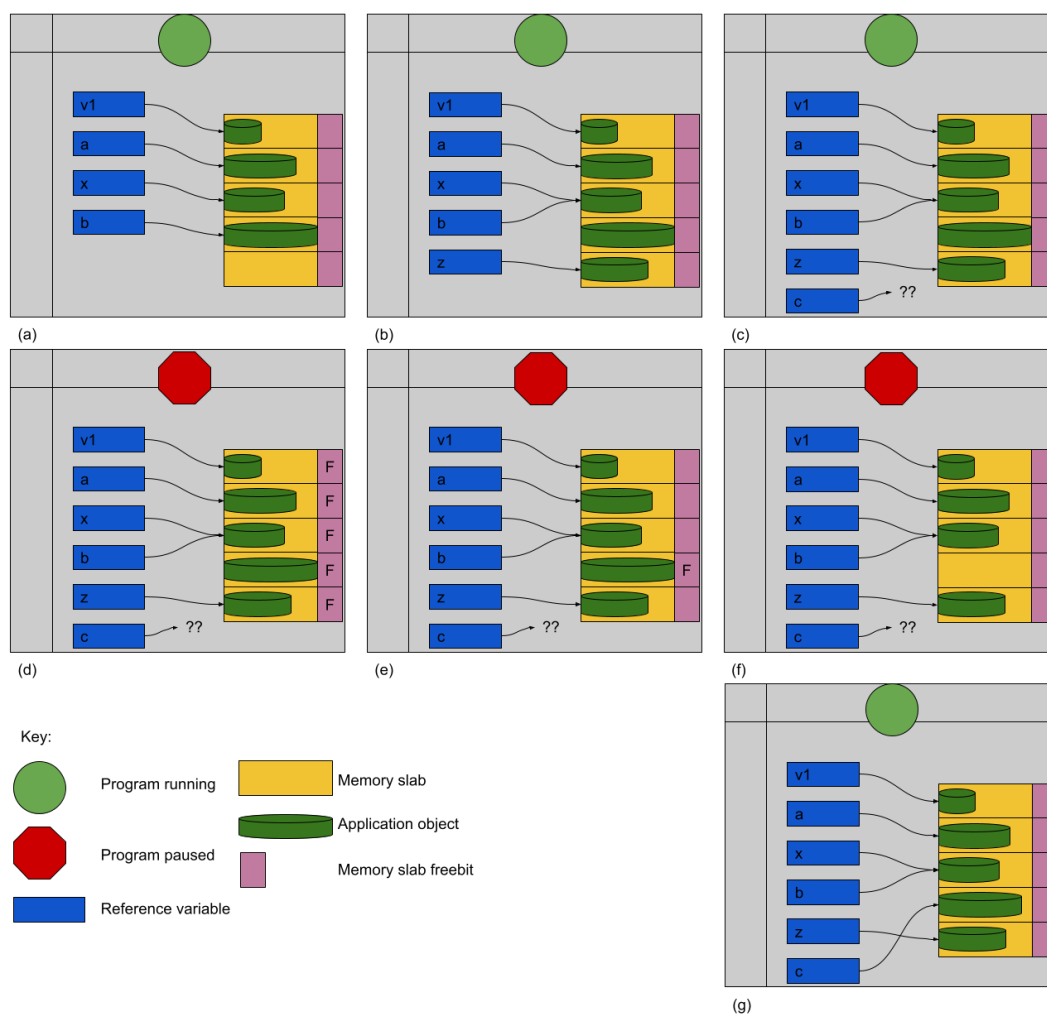
The fixed sizing of blocks in a GC'd system makes it easy/fast to walk through every block of memory. We will see shortly that the GC algorithm requires such an operation every time that it stops the program to do a cleanup. Without a consistent size, traversing the memory blocks would require that each block hold a tag indicating its size -- a waste of space and the cause of an additional memory read -- so that the algorithm could dynamically calculate the starting address of the next block.

When the programmer requests an allocation that cannot be satisfied, the programming language stops the execution of the program and does a garbage collection. The classic GC algorithm is called mark and sweep and has three steps:

1. Every block of memory is marked as free using a free bit attached to the block. Of course, this is only true of some of the blocks, but the GC is optimistic!
2. All pointers active at the time the program is paused are traced to their targets. The free bits of those target blocks are reset.
3. The blocks that are not marked as in use are released

The process is shown visually below:

Memory Management with Garbage Collection



At times (a), (b) and (c), the programmer is allocating and manipulating references to dynamically allocated memory. At time (c), the allocation request for variable **z** cannot be satisfied because there are no available blocks. A GC pause starts at time (d) and the mark-and-sweep algorithm commences by setting the free bit of every block. At time (e) the pointers are traced and the appropriate free bits are cleared. At time (f) the memory is released from the unused block and its free bit, too, is reset. At time (g) the allocation for variable **z** can be satisfied, the GC pause completes and the programming language restarts execution of the program.

This process seems great, just like reference counting seemed great. However, there is a significant problem: The programmer cannot predict when GC pauses will occur and the programmer cannot predict how long those pauses will take. A GC pause is completely initiated by the programming language and (usually) completely beyond the control of the programmer. Such random pauses of program execution could be extremely harmful to a system that is controlling a system that needs to keep up with interactions from the outside world. For instance, it would be totally unacceptable for an autopilot system to take an extremely long GC pause as it calculates

the heading needed to land a plane. There are myriad other systems where pauses are inappropriate.

The mark-and-sweep algorithm described above is extremely naive and GC algorithms are the subject of intense research. Languages like go and Java manage memory with a GC and their algorithms are incredibly sophisticated.