Object Orientation

1. What's Object Oriented Programming?

Object Oriented programming is a paradigm built off of the decomposition of an application into objects. Objects represent abstractions.

Objects are an abstraction that is encapsulated. An object can do things for you by invoking a method on it. Messages and information can be passed between objects to accomplish things.

In an Object Oriented System you build your system from a series of objects, and these objects provide you services.

2. Pillars of the Paradigm

There are four main pillars of OOP:

- **Abstraction**
- **Encapsulation**
- **Inheritance**
- **Polymorphism**

Abstraction

We use abstraction for communicating and dealing with complexity. Abstraction is how we represent details of our application.

- To abstract is to purposefully generalize details or attributes.
- We hide complexity by representing it through models.

Encapsulation

Encapsulation is the separation of what you do from how you do it. **It is a separation from the interface from the implementation.**

- Where you publish your object's capabilities to the outside world, and then you hide the details of how it's implemented.
- It is not about making a variable private, it is hiding implementation.
- **Reduces coupling**, rather than depending on the specific details of your code's implementation a user depends on what you provide. When you change your code, the

user will not be affected by it.

Inheritance

Inheritance is when a class can inherit from another class. When a class inherits from another class, that class is able to derive all the behavior of the base class.

> Inheritance is quite overrated. It is not as important to achieve reusability of code by using inheritance. There are languages that can provide you a better way of yielding reusability than inheritance.

Inheritance is useful when you want to promote **substitutability**. When you want to use an instance of a class in place of an instance of another class.

If you don't want to use substitutability, and rather use the methods of a base class in a derived class then Inheritance may not be the best choice. In this case it may be better to use ***delegation*** than inheritance. Let's take a look in some code below!

```
class Eraser {
        public void erase() { System.out.println("erasing..."); }
}

class BetterEraser extends Eraser {
        public void erase() { System.out.println("erasing efficiently...") }
}

public class Sample {
        public void eraseBoard(Eraser eraser) {
                eraser.erase();
        }

        public static void main(String[] args) {
                Eraser eraser = new Eraser();

                Sample instance = new Sample();
                instance.eraseBoard(eraser);

                BetterEraser betterEraser = new BetterEraser();
                instance.eraseBoard(betterEraser);
        }
 }
```

This code will not compile if `BetterEraser` does not inherit from `Eraser`. This is to allow us to be able to use an instance of `BetterEraser` in place of where an instance of `Eraser` would be

used.

How about we inherit another class from `Eraser`, we'll create the `Instructor` class and see what happens.

```
class Eraser {
        public void erase() { System.out.println("erasing..."); }
}

class BetterEraser extends Eraser {
        public void erase() { System.out.println("erasing efficiently...") }
}

class Instructor extends Eraser {
}

public class Sample {
        public void eraseBoard(Eraser eraser) {
                eraser.erase();
        }

        public static void main(String[] args) {
                Eraser eraser = new Eraser();

                Sample instance = new Sample();
                instance.eraseBoard(eraser);

                BetterEraser betterEraser = new BetterEraser();
                instance.eraseBoard(betterEraser);

                Instructor instructor = new Instructor();
                instructor.erase();
        }
 }
```

An unfortunate consequence of this inheritance is now that you're able to send an `Instructor` object into a method where an `Eraser` is expected. You can now treat an `Instructor` as an `Eraser` because of inheritance. We don't want the `Instructor` to be treated as an `Eraser` but rather be able to use an `Eraser`.

So what can we do to make that happen?

In Java, you can write those methods inside of the `Instructor` class and you can hold an instance of the `Eraser` class in there.

OR..... you can *replace inheritance with delegation*.

```
class Eraser {
        public void erase() { System.out.println("erasing..."); }
}

class BetterEraser extends Eraser {
        public void erase() { System.out.println("erasing efficiently...") }
}

class Instructor extends Eraser {
        private final Eraser _eraser = new Eraser();

        public void erase(){
                _eraser.erase();
        }
}

public class Sample {
        public void eraseBoard(Eraser eraser) {
                eraser.erase();
        }

        public static void main(String[] args) {
                Eraser eraser = new Eraser();

                Sample instance = new Sample();
                instance.eraseBoard(eraser);

                BetterEraser betterEraser = new BetterEraser();
                instance.eraseBoard(betterEraser);

        }
 }
```

If you want an instance of a class B to be treated as an instance of a class A then use inheritance.

However if you simply want an instance of class B to use an instance of class A, then use delegation.

- Can also give you reusability as well.

Why do people use inheritance more often than delegation?

- It's often easier to write inheritance than delegation.
    - Let's say we add a method to the `Eraser` class and we want that method in the `Instructor`, we'd have to come and change the `Instructor` class.

In this example from Java, delegation is a poor design. Why? Because this class fails the Open-Closed Principle.

> **Open-Closed Principle** - Objects or entities should be open for extension but closed for modification.

But this does not have to be the case! Let's take a look at an example from Groovy:

```
class Eraser {
        public void erase() { System.out.println("erasing...") }
}

class Instructor {
        @Delegate Eraser eraser = new Eraser()
}

instructor = new Instructor()
instructor.erase()
```

There is a `@Delegate` which is the delegation mechanism that Groovy provides us to use.

- The delegation in Groovy is OCP compliant.

As you can see from this example, not all languages come with delegation mechanisms.

Polymorphism

Polymorphism says go ahead and call a method, and the method that's called is based on the type of the object at run-time and not the type of the object at compile-time.

An example is that in the `eraseBoard` method, we can invoke the `erase` method on either an `Eraser` or a `BetterEraser` at runtime. It is not based on the type of the `Eraser` that you've specified at compile-time. So this is polymorphic.

3. Classes, Instances, Fields, Properties, and Methods

1. You can create ***instances*** of classes, except for an abstract class in most languages.
2. Classes can have ***fields***, they may have ***properties*** which you can call upon an object to implement properties.

3. Classes may have ***methods***, some are modifiers, some are query methods, you may have final methods which shouldn't be overridden, or you may have virtual functions.

Singletons

A singleton is an object that can have only one instance.

> Singletons are a problem in Java and C#

In this code below, we create a singleton. If an instance is already created then we return that instance. However if the instance is null we create that new `Singleton`.

```
class Singleton{
        private static Singleton instance;

        public static Singleton getInstance() {
                if (instance == null) {
                        instance = new Singleton();
                }
                return instance;
        }
}

public class Sample {
        public static void main(String[] args) {
                Singleton inst1 = Singleton.getInstance();
                Singleton inst2 = Singleton.getInstance();
        }

}
```

However, there are multiple ways to break this code!!!

- **Reflection** - You can then go to the `Singleton` class, get a reference to the constructor, and use the constructor info object to create an instance. Like in C#.
- **Multi-threading** - This code is not thread safe, under a multithreaded system, you could have multiple threads calling this code at the same time. Both threads check the instance at the same time and both see null and create an instance.

Your code may appear to be correct when you write a singleton, but it can very easily be problematic. Creating a thing as a singleton in Java is really difficult.

The issue of singletons was so major in Java, that Scala went ahead an fixed the problem themselves.

```
object Singleton {
        def op1() { println("op1.....") }
}

Singleton.op1
```

In the case of Scala we create singletons with the keyword `object`.

- notice we didn't use the keyword `new` in any of this code.

4. Static

Static members and methods aren't linked to any object instance and can be accessed directly by the class reference.

> A static member contains the same value across all class variables. A static method or field applies to every object of the class, instead of being associated with just the declared variable of the class.

> In Java, Static methods (or fields) are only allowed to access other static methods and fields within its defined class.

"Static is evil" - Venkat

In an Object Oriented project, you know that you have objects and methods reside on objects. But what happens with the static?

- When you write a class you write methods that belong to the instance and then methods that belong to a class. And a whole bunch of confusion

Statics aren't fun to work with while testing, while developing a web app or even in an OO application.

- It's pretty difficult to mock static methods.
- Requests to a web app become convoluted with statics are used.

Scala and Static

Scala is a fully Object Oriented language that doesn't support static. Scala instead supports companion objects. You can create singletons that accompany another class as a companion.

```
class Marker(val color : String) {
        println("Creating a marker of color " + color)
```

```
        override def toString = {
                "Marker color is " + color
        }
}

new Marker("blue")
new Marker("blue")
new Marker("red")
```

```
Creating a marker of color blue
Creating a marker of color blue
Creating a marker of color red
```

We now have multiple instances of marker, it's no long a singleton. We have two blue markers with two different instances. We want there to be a single instance for a marker of the same color.

```
class Marker(val color : String) {
        println("Creating a marker of color " + color)
        override def toString = {
                "Marker color is " + color
        }
}

object MarkerFactory {
        val markers = Map(
        "blue" -> new Marker("blue"),
        "red" -> new Marker("red"),
        "green" -> new Marker("green")
        )

        def getMarker(color : String) = {
                markers.get(color)
        }
}

val blueMarker = MarkerFactory.getMarker("blue")
println(blueMarker)
```

Notice that `MarkerFactory` is defined as a an object, a singleton, and as a result `getMarker` is a method that you can directly call on that object.

- So in Scala you implement *static methods* not as a static methods but as methods on singletons.

So when we run this code, Scala returns `Some(Marker color is blue)` from our `getMarker("blue")` call. We get an object of an option type to avoid a null pointer exception.

- Returned either a `None` object or a a `Some` object.

To get the real object itself you can add `.getOrElse(null)` after our `getMarker()` call.

You can also use a companion object here.

```
class Marker(val color : String) {
        println("Creating a marker of color " + color)
        override def toString = {
                "Marker color is " + color
        }
}

object Marker {
        val markers = Map(
        "blue" -> new Marker("blue"),
        "red" -> new Marker("red"),
        "green" -> new Marker("green")
        )

        def getMarker(color : String) = {
                markers.get(color)
        }
}

val blueMarker = MarkerFactory.getMarker("blue")
println(blueMarker.getOrElse(null))
```

The singleton has the same name as the class, so now it's a companion class. You are only now allowed to us any methods of the `Marker` class on your singleton.

- The companion object has full access to the class that it is a companion to.

Scala being a purely OO language means that every method belongs to an instance.

- Some belong to a method of a class, some belong to a singleton.

5. Aggregation vs Composition

Let's jump into an example in C++.

- In C++, you have **association** - a relationship between two objects. You also have **aggregation** where one object owns another object. And you also have **composition**, where one object is composed of another object.

These ideas are much clearer in C++.

> Say you create a class `Person`. There is memory allocated for a `Person`. And embedded in that memory is data for the objects related to it. You can find a `Heart` and a `Brain` in that memory.

```
class Person {
        Heart* pHeart;
        Brain theBrain;
}
```

Brain is 12 byes in size
Pointer to Heart is 4 bytes.
Person is of size 12 + 4 + ...

In this case you would say a brain is a composition and a heart is an aggregation.

- Why is this?
  - The brain is embedded into the `Person` object. Each `Person` object is composed of a `Brain` object.
  - While the `Heart` is just being pointed to, this `Person` owns a heart via a pointer.

Now imagine the consequences of this. When the brain is embedded into this `Person` object. When you delete the `Person` object, you also delete the `Brain`. But the `Heart` is not, just the pointer. You can change the `Heart` object and you can reset the pointer to a pointer of another object.

When it comes to Aggregation you can change the object that you aggregate. But in Composition you ***CANNOT***.

In composition the memory of the object is embedded within another. The lifetime of the objects are tied together.

In Java and C#, there is no distinction. Both instances of `Heart` and `Brain` are pointers. So you can point them to something and change them later on.

```
class Person {
        Heart heart;
        Brain brain;
}
```

Java and C# do not support composition by default. But you can still model it. So how do we do that?

```
class Person {
        private Heart heart;
        private Brain brain;

        public Person(Brain theBrain, Heart aHeart) {
                brain = theBrain;
                heart = aHeart
        }

        public void changeHeart(Heart aHeart) {
                heart = aHeart;
        }
}
```

Notice that you have not provided a way to change the `Brain` in the same way you can change the `Heart`.

From the point of view of object modelling, the concepts of association, composition and aggregation have a distinct meaning. But from the point of view of coding, you'll have to pick and choose how to express these concepts based on the nature of the language you are using.

6. Interfaces

In C++ you have no concept of interfaces, you have abstract classes and purely abstract classes.

- A purely abstract class is a class made up entirely of purely virtual methods.
  So there is no distinct concept of interfaces although it is there in other forms.

In Java there is a clear distinction of interfaces. As there is in C# and Groovy.

- An **interface** is a grouping of methods with no implementation.

However, languages like Scala and Ruby do not provide interfaces. Because interfaces are traditionally used for **designed by contract**.

- So you're promising that you abide by a certain implementation when using an interface.

Certain languages prefer a design by capability instead of a design by contract. Certain design favors influence the availability of interfaces.

- So it depends on what language you are using and what design approach you are following.

Interfaces define a certain method and because this method is expected you can verify it at compile time and at runtime you know the right method will be called because in addition to the interface you can rely on ***polymorphism***.

Interface Collision

What if you have two interfaces with exactly the same method? *Trouble awaits in some languages...*

> In this example, you will likely say "Hey, but you're the one naming these methods why don't you avoid the collision by naming them differently". Well let's pretend that this is a real life example. What would happen in the case where you import two different libraries with two interfaces that have methods of the same name?

```
interface FootballPlayer {
        void play();
}

interface PianoPlayer {
        void play();
}

class FootballPlayingPerson implements FootballPlayer {
        public void play() {
                System.out.println("kick the ball...");
        }
}

class PianoPlayingPerson implements PianoPlayer {
        public void play() {
                System.out.println("press keys...")
        }

}

class Sample {
        public static void playFootball(FootballPlayer player) {
```

```
                player.play();
        }

        public static void playPiano(PianoPlayer player) {
                player.play();
        }
        public static void main(String[] args) {
                FootballPlayingPerson peter = new FootballPlayingPerson();

                playFootball(peter);

                PianoPlayingPerson susan = new PianoPlayingPerson();
                playPiano(susan)
        }
}
```

So far so good, there are no issues or collisions here. But what if we add a new class called `MultiSkilledPerson`?

```
interface FootballPlayer {
        void play();
}

interface PianoPlayer {
        void play();
}

class FootballPlayingPerson implements FootballPlayer {
        public void play() {
                System.out.println("kick the ball...");
        }
}

class PianoPlayingPerson implements PianoPlayer {
        public void play() {
                System.out.println("press keys...")
        }

}

class MultiSkilledPerson implements PianoPlayer, FootballPlayer {
        public void play() {
                System.out.println("????")
        }
}
```

```
class Sample {
        public static void playFootball(FootballPlayer player) {
                player.play();
        }

        public static void playPiano(PianoPlayer player) {
                player.play();
        }
        public static void main(String[] args) {
                FootballPlayingPerson peter = new FootballPlayingPerson();

                playFootball(peter);

                PianoPlayingPerson susan = new PianoPlayingPerson();
                playPiano(susan)

                MultiSkilledPerson bob = new MultiSkilledPerson();
                playFootball(bob);
                playPiano(bob);
        }
}
```

Now how do we distinguish whether we called `play()` as a `PianoPlayer` or as a `FootballPlayer` in the case of our `MultiSkilledPerson`? Java provides no context nor distinction. But C# does!

```
using System;
using System.Collections.Generic;
using System.Ling;
using System.Text;

namespace Sample {
        interface FootballPlayer {
                void Play();
        }

        interface PianoPlayer {
                void Play();
        }

        class FootballPlayingPerson : FootballPlayer {
                public void Play() {
                        Console.WriteLine("kick the ball...");
                }
```

```
        }

        class PianoPlayingPerson : PianoPlayer {
                public void Play() {
                        Console.WriteLine("press key...");
                }
        }

        class MultiSkilledPerson : PianoPlayer, FootballPlayer {
                public void Play() {
                        Console.WriteLine("press keys...");
                }

                void FootballPlayer.Play() { // explcit interface!
                        Console.WriteLin("kick the ball...")
                }
        }

        class Program {
                static void playFootball(FootballPlayer player) {
                        player.Play();
                }

                static void Main(string[] args) {
                        FootballPlayingPerson peter = new FootballPlayingPerson();
                        PlayFootball(peter);

                        PianoPlayingPerson susan = new PianoPlayingPerson();
                        PlayPiano(susan);

                        MultiSkilledPerson bob = new MultiSkilledPerson();

                        PlayFootball(bob);
                        PlayPiano(bob);
                }
        }
}
```

C# supports explicit interfaces to provide context. In this case we used an explicit interface for `MultiSkilledPerson`. When we invoke the `play` function through a `FootballPlayer` function call on a `MultiSkilledPerson` object, then we will call the appropriate method.

7. Overloading

Overloading is a way for you to name multiple methods by the same name.

- For example you can `pay()` by a cheque, you may `pay()` by a credit card or debit card, or maybe by cash.
- Back in time C would make you name each method by a different name however C++ has introduced function overloading.

In the terms of binding, overloading is ***static binding***. Meaning that it determines what method to call at compile time.

Overloading requires that both functions are within the same scope.

- If you have a method in two different classes of the same name, then this is **NOT** overloading as they are in differing scopes.

> Overloading is confused with Polymorphism but overloading does not provide any extensibility to code in any way.

> Fun fact! **Ruby does not support method overloading.** When a second method is defined with the same name it completely overrides the previously existing method!.

8. Polymorphism

Polymorphism is an extremely important concept that provides extensibility in code.

Methods are **polymorphic** when they have the same name but are in different scopes.

Polymorphism in a statically typed language: The methods that you talk about have the same name but in two different scopes that are related as a base class and derived class.

Polymorphism in a dynamically typed language: The two methods can be any class scope, it doesn't matter if there's a relationship between the two classes.

The method that gets called is based on the type of the object that you invoking the method on at runtime.

You may have the impression that inheritance is required in order to have polymorphism. However, this may not be the case in all languages. This is what you would think if you have only used **statically typed languages**.

What Polymorphism does requires is the ***presence of the method on that object at runtime*** when you call it.

- It does not require an interface

However, because of the restrictions of statically typed languages Polymorphism can lead to odd behaviors at times.

- For example in C++, it may be a crime to override a function that is not virtual.
- In Java and C#, you need to take effort to preserve the parameters that polymorphic methods take.

Let's take a look at an example!

```
abstract class Animal {
        public final void eat() { System.out.println("eating..."); }
        public abstract void makeNoise();
}

class Dog extends Animal {
        @Override public void makeNoise() {
                System.out.println("bark...");
        }
}

class Cat extends Animal {
        @Override public void makeNoise() {
                System.out.prinln("meow...");
        }
}

class Sample {
        public void playWithAnimal(Animal animal) {
                animal.eat();
                animal.makeNoise();
        }

        public static void main(String[] args) {
                playWithAnimal(new Dog());
                playWithAnimal(new Cat());
        }
}
```

We know that we can't call a method on an abstract class because there's no implementation.

- However the compile will let us run it anyways, because Java knows that at runtime that the `Animal` object that you pass to the `playWithAnimal` method will be an instance of another class that has the methods of `Animal`.

What Java is going to do here is that it's going to call the `makeNoise` function as a polymorphic function call.

- Java is simply going to treat the method polymorphically.

> Static methods are NEVER polymorphic. Statics and Polymorphism are enemies.

Notice that we call the same function with two different objects. And that our polymorphic functions are *dynamically bound*.

What's the difference between static and dynamic binding?

In **static binding**, the compiler clearly knows which method it's going to call at compile time. In the case of **dynamic binding**, the compiler defers to runtime to decide on which method to call.

- Polymorphism relies on runtime binding to know which function call.
  - We don't know that our noise will be a bark until a `Dog` is passed at runtime.

```
import java.util.ArrayList;

class Sample {
        public static void main(String[] args) {
                use1();
                use2();
        }

        private static void use1() {
                ArrayList<Integer> list = new ArrayList<Integer>();
                list.add(1);
                list.add(2);

                System.out.println("Number of elements is" + list.size());

                list.remove(0);
                System.out.println("Number of elements is" + list.size());
        }

        private static void use2() {
                Collection<Integer> list = new ArrayList<Integer>();
                list.add(1);
                list.add(2);

                System.out.println("Number of elements is" + list.size());
```

```
                list.remove(0);
                System.out.println("Number of elements is" + list.size());
        }
}
```

Notice how you make only one change in the code, you are treating the `ArrayList` as an `ArrayList` versus treating an `ArrayList` as a `Collection` (it's base class).

- You would argue that there is no change to this code at all.

However, both functions are producing different outputs from one another.

- Notice that the `remove` method in `use2`, it isn't being called on an instance of `ArrayList` it is being called on an instance of the `Collection` interface.
  - The `remove` method on a `Collection` removes an `Object` and not an `int`!!!
  - Java's API was designed very wrong. They overrode the base class but changed the signature on the method.

This here is called auto-boxing. It takes the `0` and instead of treating it as an `int`, it treats it as an `Integer` object and passes it to the remove method. Once the `Integer` arrives to the `remove` function there is no object 0 so instead of removing it, it ignores it!

What polymorphism should do is that when you call a method on an object, it shouldn't matter how you called that method **the effect should be exactly the same**.

- It shouldn't matter whether you called it as a base or derived class, the behavior should be the same.

```
class CurrencyConverter {
        public void convert(double value) {
                System.out.println("CurrencyConverter's convert(double) called...")
        }
}

class BetterCurrencyConverter extends CurrencyConverter {
        public void convert(int value) {
                System.out.println("BetterCurrencyConverter's convert(int) ")
        }
}

class Sample {
        public static void useConverter(CurrencyConverter converter) {
                converter.convert(5);
        }
```

```
        public static void main(String[] args) {
                useConverter(new CurrencyConverter());
                useConverter(new BetterCurrencyConverter());
        }
}
```

If polymorphism worked correctly, it should know to call the method in the derived class!

Line 9 causes issues because we changed the signature of the `convert` function; there we are expecting an `int` instead of a `double`.

- We passed an integer, yet we called the base class's method which was expecting a double.

What happened was that when you called the `convert` method, it didn't matter if you passed a `CurrencyConverter` or a `BetterCurrencyConverter`, this call ended up calling the method on the `CurrencyConverter` alone, on its own.

This was still dynamic binding, it checked the type of the `converter` object. So then how did it not call the function for the `BetterCurrency Converter`?

Well this is because what's happening is that we:

1. Postpone to runtime the method to call
2. At runtime, examine the real object that converter refers (or points) to
3. Call the method `convert` on it.

Spot the sneakily hidden data type conversion in Java

Remember that Java is a **statically typed language**. At compile time we have to undergo **type verification**.

- The compiler performs a type verification that asks if we can pass a certain type to the method.
    - HOWEVER, recall that at ***compile time***, the type of `converter` is `CurrencyConverter`!

When we pass a `5`, the compiler will implicitly convert the data type of the integer to a double.

- So before it postpones the method call to runtime, it has **already changed the data type at compile time***.
- At runtime, when the `BetterCurrencyConverter` is called, the `convert` function that takes an `int` is not suitable for our `double`. So then the `BetterCurrencyConverter` then takes us to the base class.

- Because instead of overriding the `convert` function, we have overloaded the `convert` function! Because we have two different signatures in differing scopes but under the same name.

In a dynamically typed language, type verifications do not happen at compile time, they are postponed at runtime.

- As a result, the runtime behavior determines the call.

9. Multimethods

The ability for dynamically typed languages to postpone type conversions to runtime is why you have multi-methods in dynamic languages.

Multimethods - the method you call is determined by the type of the target object (left of the dot) and the type of the parameters.

> Polymorphism does not consider the type of the parameters.

Multimethods can be thought of as an extension of polymorphism. They are normally only found in dynamic languages because it relies on the runtime type.

```
class CurrencyConverter {
        public void convert(double value) {
                System.out.println("CurrencyConverter's convert(double) called...")
        }
}

class BetterCurrencyConverter extends CurrencyConverter {
        public void convert(double value) {
                System.out.println("BetterCurrencyConverter's convert(double) ")
        }
        public void convert(int value) {
                System.out.println("BetterCurrencyConverter's convert(int) ")
        }
}

public void useConverter(CurrencyConverter converter) {
        converter.convert(5);
        converter.convert(5.0);
}

useConverter(new CurrencyConverter())
useConverter(new BetterCurrencyConverter())
```

Groovy's method dispatching is quite complex. If you were to look at the bytecode produced you will notice that it calls the methods on a call-site, which does the dispatching at runtime.

- The decision of which method gets called is postponed to the runtime rather than compile time in groovy.

Let's look at another example of multimethods at work.

Recall the code from earlier...

```
import java.util.ArrayList;

class Sample {
        public static void main(String[] args) {
                use1();
                use2();
        }

        private static void use1() {
                ArrayList<Integer> list = new ArrayList<Integer>();
                list.add(1);
                list.add(2);

                System.out.println("Number of elements is" + list.size());

                list.remove(0);
                System.out.println("Number of elements is" + list.size());
        }

        private static void use2() {
                Collection<Integer> list = new ArrayList<Integer>();
                list.add(1);
                list.add(2);

                System.out.println("Number of elements is" + list.size());

                list.remove(0);
                System.out.println("Number of elements is" + list.size());
        }
}
```

The output was:

```
Number of elements is 2
Number of elements is 1
```

```
Number of elements is 2
Number of elements is 2
```

The Java API unfortunately failed to remove an element because of auto-boxing. The two different functions had a differing behavior because one treats our list as the derived class and the other treats it as a the base class.

How does this code work in Groovy?

```
import java.util.ArrayList;

class Sample {
        public static void main(String[] args) {
                use1();
                use2();
        }

        private static void use1() {
                ArrayList<Integer> list = new ArrayList<Integer>();
                list.add(1);
                list.add(2);

                System.out.println("Number of elements is" + list.size());

                list.remove(0);
                System.out.println("Number of elements is" + list.size());
        }

        private static void use2() {
                Collection<Integer> list = new ArrayList<Integer>();
                list.add(1);
                list.add(2);

                System.out.println("Number of elements is" + list.size());

                list.remove(0);
                System.out.println("Number of elements is" + list.size());
        }
}

use1()
use2()
```

```
Number of elements is 2
Number of elements is 1
Number of elements is 2
Number of elements is 1
```

Notice how we call `use1` and `use2` with the same exact code in Groovy, but because Groovy has multi-methods the two functions behavior the same way.

Ruby as an example

> In the case of Ruby, Ruby is a dynamically typed language so there are no type checks at compile time, and Ruby has no function overloading. So by default Ruby always has multi-methods.

> Or you could say that Ruby always has polymorphism because you cannot call the method of the derived classes, because Ruby simply doesn't care about the signature of the methods. However, you cannot accidentally call the base method because it is the derived method that always takes up things.

```ruby
class Base
        def foo(value)
                value + 2
        end
end

class Derived < Base
        def foo(*values)
                values.size()
        end
end

def use(inst)
        puts inst.foo(4)
end

use(Base.new)
use(Derived.new)
```

```
6
1
```

Ruby will always call the method on the derived class, it doesn't take account for the data type of the parameters in the method.

- Our one value, our one integer, was assumed to be a list when we passed it into our derived method.
  Ruby doesn't know the value that's being passed in and assumes it's type at runtime. Ruby entirely overrides this method all the time, so you can argue that Ruby does not provide multimethods.
- Because Ruby is always going to call the method on the derived class.
  - The data type is not a consequence at all when calling a method.

9. Execute Around Method Pattern

When we deal with OOP we will often need to deal with garbage collection.

- In C++, if you allocate an object on the stack it will go away on its own. If you allocate an object on the heap you will need to delete it yourself, or else it will stay in the heap occupying memory.
- In Java and C# there is automatic garbage collection. In these languages you don't have much control over the scope of the collection of this garbage, because Java has the ARM (automatic resource management).

An Execute Around Method Pattern in Scala

```
class Resource {
        println("Creating resource")
        def close() {
                println("cleaning up the resource")
        }

        def op1() { println("op1...") }
        def op2() { println("op2...") }
        def op3() { throw new RuntimeException }
}

val resource = new Resource
resource.op1
resource.op2
resource.op3
resource.close
```

Notice how we can never close the program because `op3` will always throw a `RunTimeException` before we can even invoke the `close` method.

```
class Resource {
        println("Creating resource")
        def close() {
                println("cleaning up the resource")
        }

        def op1() { println("op1...") }
        def op2() { println("op2...") }
        def op3() { throw new RuntimeException }
}

val resource = new Resource
try {
        resource.op1
        resource.op2
        resource.op
} finally {
        resource.close
}
```

Notice how when the code *blows up*, it calls the `close` function before it *blows up*.

- The code called `op1` and `op2` and then the cleanup happened before the error.

This is exactly what the ARM in Java provides. The problem here is that as a programmer we have worry about writing this out every single time. We have to worry about writing the clean up every time. Let's look into this further...

```
class Resource private {
        println("Creating resource")
        private def close() {
                println("cleaning up the resource")
        }

        def op1() { println("op1...") }
        def op2() { println("op2...") }
        def op3() { throw new RuntimeException }
}

val resource = new Resource
try {
        resource.op1
        resource.op2
        resource.op
} finally {
```

```
        resource.close
}
```

We've made the constructor private and the `close` function private.

- But then we can't create an instance of this class! Don't fret let's create a companion object and see where that takes us.

```
class Resource private {
        println("Creating resource")
        private def close() {
                println("cleaning up the resource")
        }

        def op1() { println("op1...") }
        def op2() { println("op2...") }
        def op3() { throw new RuntimeException }
}

object Resource {
        def use(block : Resource => Unit) {
                val resource = new Resource
                try {
                        resource.op1
                        resource.op2
                        resource.op
                } finally {
                        resource.close
                }
        }
}

Resource.use { resource =>
        resource.op1
        resource.op2
        resource.op3
}
```

This is called the **Execute Around Method Pattern**, where we execute a certain pre-operation and post-operation around this method.

- Execute around method patterns are available in Java and C#, but are limited.

In languages that support closures and lambda expressions you are able to use this pattern.

An Execute Around Method Pattern in Ruby

```
class Resource
        def initialize()
                puts "creating resource"
        end

        def op1
                puts "op1 called..."
        end

        def op2
                puts "op2 called..."
        end

        def close
                puts "clean up..."
        end

        private :close

        def self.use
                resource = Resource.new
                begin
                        yield resource
                ensure
                        puts "clean up resource here"
                end
        end

end

resource = Resource.new
resource.op1
resource.op2
```

Notice that the code *blew up* but before it threw that exception it was cleaned up as we wanted.

An Execute Around Method Pattern in Groovy

```
class Resource {
        def static use(closure) {
                Resource resource = new Resource()
                println "create resource"
```

```
                try {
                        resource.with closure
                } finally {
                        println "clean up..."
                }
        }

        def op1() { println "op1 called..." }
        def op2() { println "op2 called..." }
}

Resource.use {
        op1()
        op2()
}
```

Groovy provide some very concise code that allows us to create a cloned closure, to delegate it and call the clone on an invisible object.

11. Traits and Mixins

Recall any issues from multiple inheritance?

1. You'll have to deal with method collision. Like in the case of interface collision seen earlier.
2. Inheriting from two base classes who share a parent class.
    - Note; Java and C# does not support multiple inheritance.

To avoid multiple inheritance you can use traits and mixins.

- Traits are a concept in Scala, which can be used at compile time
- Mixins are a concept in Ruby and Groovy which can be used at run time.

Traits

```
trait Friend {
        val name : String
        def listen = println("I am " + name + " your friend, listening...")
}

def talkToAFriend(friend : Friend) = {
        friend.listen
}
```

```
class Human(val name : String) extends Friend {}

class Animal(val name2 : String) {}
class Dog(override val name : String) extends Animal(name2) with Friend

val peter = new Human("Peter")
peter.listen
talktToAFriend(peter)

val peter = new Dog("Snowy")
snowy.listen
talktToAFriend(snowy)
```

> A **Trait** is an interface with partial implementations in it.

In this example our `Dog` class inherits from the `Animal` class and from the `Friend` trait.

- This is an example of a ***mixin***, what we did was we extracted the traits separately and mixed it into the class.

Scala does not only allow you to mix in traits into classes, you can also mix in traits at the object level.

```
trait Friend {
        val name : String
        def listen = println("I am " + name + " your friend, listening...")
}

def talkToAFriend(friend : Friend) = {
        friend.listen
}

class Human(val name : String) extends Friend {}

class Animal(val name2 : String) {}
class Dog(override val name : String) extends Animal(name2) with Friend

val peter = new Human("Peter")
peter.listen
talktToAFriend(peter)

val peter = new Dog("Snowy")
snowy.listen
talktToAFriend(snowy)
```

```
class Cat(override val name : String) extends Animal(name)

val alf = new Cat("Alf")

// mixing in a trait at the object level
val yourCat = new Cat("yourcat") with Friend
yourCaat.listen
talkToAFriend(yourCat)
```

Specific instances of a class can be mixed in with traits using the same syntax, in Scala.

Ruby

```
module Friend
        def listen
                puts "I am " + name + " your friend... listening"
        end
end

class Human
        include Friend
        attr_accessor :name
end

class Dog
        attr_accessor :name
end

class Cat
        attr_accesor :name
end

peter = Human.new
peter.name = "Peter"

peter.listen

snowy = Dog.new
snowy.name = "Snowy"
snowy.listen

alf = Cat.new
alf.name = "Alf"

your_cat = Cat.new
```

```
class <<your_cat
        include Friend
end

your_cat.name = "yourCat"
your_cat.listen
```

You are able to mix in traits not only into classes but objects as well at runtime in Ruby.

> What Ruby does here is that it chains these objects together. But we can look at that again later.

What's nice about traits is that they don't follow a vertical inheritance hierarchy.

Here's what happens in Ruby:

- `alf` is a pointer to a `Cat` instance
- If the `Cat` instance does not have the method then check the `Cat` metaclass to see if it has the method.
  Because in Ruby, instances of a class can have methods that the metaclass does not have.

`your_cat` is a bit different than `alf` here

- `your_cat` first checks if the *virtual class* has the method. The virtual class has the methods of the `Friend` mixin mixed in.
- Then it checks the instance
- Then it checks the metaclass

```
module Mixin1
        def f1()
                puts "Mixin1 f1 called"
        end
        def g1()
                puts "Mixin1 g1 called"
        end
end

module Mixin2
        def f2()
                puts "Mixin 2 f2 called"
        end
        def g1()
```

```
                puts "Mixin2 g1 called"
        end
end

class MyClass
        include Mixin1
        include Mixin2
        def g1()
                puts "Mixin2 g1 called"
        end
end

inst = MyClass.new
inst.f1
inst.f2
inst.g1
```

Chaining order of our mixins

`inst -> (vc Myxlass) -> (vc Mixin2) -> (vc Mixin1) -> MyClass metaclass`

Rather than a vertical hierarchy it's a right to left hierarchy.

```
abstract class Writer {
        def write(msg : String)
}

class StringWriter extends Writer {
        val target = new StringBuilder
        override def write(msg : String) = {
                target.append(msg)
        }
}

def writeStuff(writer : Writer) = {
        writer.write("This is stupid")
        println(writer)
}

trait UpperCaseFilter extends Writer {
        abstract override def write(msg : String) = {
                super.write(msg.toUpperCase())
        }
}

trait ProfanityFilter extends Writer {
        abstract override def write(msg : String) = {
```

```
                super.write(msg.replace("stupid", "s****"))
        }
}

writeStuff(new StringWriter) // instance of StringWriter

writeStuff(new StringWriter with UpperCaseFilter)
// instance of an anonymous class

writeStuff(new StringWriter with ProfanityFilter)
//notice that order matters
writeStuff(new StringWriter with UpperCaseFilter with ProfanityFilter)
writeStuff(new StringWriter with ProfanityFilter with UpperCaseFilter)
```

The `super` doesn't mean that you're calling on the base class. Notice that the base class `Writer` is abstract and doesn't even have a `write` function that we can call.

What `super` means is that you're calling the class that the `UpperCaseFilter` trait is attached to. Or in the class that the instance is attached to.

Line 25 is not an instance of `StringWriter`! It is an instance of an anonymous class that extends from `StringWriter` and mixes in `UpperCaseFilter`.

- So the sequence is:
  StringWriter <- UpperCaseFilter <- anonymous class <- instance

Notice that the order matters. In the case of our first call that mixes in `ProfanityFilter`, the "STUPID" gets replaced with "S****". *While in our second call this is not the case*, the word "STUPID" is not replaced. Because `ProfanityFilter` only works in lowercase.

- This is because of the order of function calls in the right to left sequencing.

Let's try to recreate this example in Ruby

```
module UpperCaseFilter
        def write(msg)
                super msg.update
        end
end

class StringWriter
        attr_reader :target
        def initialize
                @target = ''
        end
```

```
        def write(msg)
                @target << msg
        end
end

def write_stuff(writer)
        writer.write("This is stupid")
        puts writer.target
end

def write_stuff(writer)
        writer.write("This is stupid")
        puts writer.target
end

write_stuff(StringWriter.new)
write_stuff(StringWriter.new.extend(UpperCaseFilter))
write_stuff(StringWriter.new.extend(ProfanityFilter))
```

> Here in Ruby this is a runtime injection instead of a compile time manipulation that Scala does.

12. Covariance and Contravariance

Typically when you create objects or values in a strong type system, it will enforce the types at compile time.

Covariance is where you can treat an object of that type or as an object of its base class.

Contravariance is where you can treat an object of a base class as an object of one of its derived classes.

An example of a Covariance in Java:

```
class Animal {}
class Cow extends Animal {}

class Barn {
        Animal getAnimal() { return null; }
}

class BarnOfCow extends Barn {
        @Override Cow getAnimal() { return null; }
}
```

```
public class Sample {
        public static void main(String[] args)
                String str1 = "hello"
        Object obj = str1;
}
```

On line 9, the base method `getAnimal` is returning an `Animal` while the derived method is returning a `Cow`.

However Java is careful of ***contravariance***. We could not replace `Cow` with `Animal` on this line as we would be returning an "incompatible type".

```
class Animal {}

public class Sample {
        public static void playWithANimal(List<Animal> animals) {}
        public static void copy(List<Animal> animals) {}

        public satic void main(String[] args) {
                List<Animal> animals1 = new ArrayList<Animal>();
                List<Dog> dogs1 = new ArrayList<Dog>();
                List<Dog> dogs2 = new ArrayList<Dog>();

                playWithAnimal(animals1)

                List<Animal> animals2 = new ArrayList<Animal>();

                copy(animals1, animals2)

                copy(dogs1, dogs2) //an attempt at covariance. ERROR!

                copy(animals1, dogs1) //ERROR!

                copy(dogs1, animals1) //ERROR! AGAIN, same issue
        }
}
```

Java is saying you can't use covariance on this type, you can't call copy by sending a list of `Dog` objects when you need a list of `Animal` objects.

- If you were to send a collection of dogs to copy a collection of animals then you may end up putting the wrong type of animal into this list. Like for example a crocodile or a tiger.

```
class Animal {}

public class Sample {
        public static void playWithANimal(List<Animal> animals) {}
        public static void <T> copy(List<T> animalssrc, List<T> animalsdest) {}

        public satic void main(String[] args) {
                List<Animal> animals1 = new ArrayList<Animal>();
                List<Dog> dogs1 = new ArrayList<Dog>();
                List<Dog> dogs2 = new ArrayList<Dog>();

                playWithAnimal(animals1);

                List<Animal> animals2 = new ArrayList<Animal>();

                copy(animals1, animals2);

                copy(dogs1, dogs2); //an attempt at covariance. ERROR!

                copy(animals1, dogs1); //ERROR!

                copy(dogs1, animals1); //ERROR!
        }
}
```

The Java compiler tries to protect you. It never wants you to treat a collection of derived types as a collection of base types. And vice versa.

There are two ways to specify the type in this code. To allow the previous copy statements.

```
class Animal {}

public class Sample {
        public static void playWithANimal(List<Animal> animals) {}
        public static void <T> copy(List<T> animalssrc, List<? super T>
animalsdest) {}

        public satic void main(String[] args) {
                List<Animal> animals1 = new ArrayList<Animal>();
                List<Dog> dogs1 = new ArrayList<Dog>();
                List<Dog> dogs2 = new ArrayList<Dog>();

                playWithAnimal(animals1);

                List<Animal> animals2 = new ArrayList<Animal>();
```

```
                copy(animals1, animals2);

                copy(dogs1, dogs2); //an attempt at covariance. ERROR!

                copy(animals1, dogs1); //ERROR!

                copy(dogs1, animals1); //ERROR!
        }
}
```

In this case, we are specifying that the `animalsdest` is any type that is a superclass of the type `T` or an instance of `T` itself.

- This can be a list of its type or a list of its base type. --> **Contravariance**

Abstract Types

Do not confuse abstract types with abstract base classes.
An **abstract type** is where a particular type is not decided until a later time.

- A class will consider a type as abstract itself and let a derived class define what type it is.

```
abstract class Fruit() { }
class Apple extends Fruit { }
class Orange extends Fruit { }

abstract class Basket {
        void put(Fruit fruit) {
                System.out.println("put a fruit into a basket")
        }
}

class BasketOfApples extends Basket {
        void put(Apple fruit) {
                System.out.println("put a fruit into a basket of apples")
        }
}

class BasketOfOranges extends Basket {
        void put(Orange fruit) {
                System.out.println("put a fruit into a basket of oranges")
        }
}
```

```
public class Sample {
        public static void putFruit(Basket basket, Fruit aFruit) {
                System.out.println(aFruit)
        }

        public static void main(String[] args) {
                putFruit(new BasketOfApples(), new Apple());
                putFruit(new BasketOfOranges(), new Orange)
        }
}
```

If you don't override the `put Orange` method then it is hiding the `put Fruit` method. What will happen is that it will call the base class based on the type matching.

- We aren't overriding the `put` method in our `BasketOfApples` and `BasketOfOranges` classes we're just hiding this method.
  One way we can avoid this is by using `instanceof` as seen below:

```
abstract class Fruit() { }
class Apple extends Fruit { }
class Orange extends Fruit { }

abstract class Basket {
        void put(Fruit fruit) {
                System.out.println("put a fruit into a basket")
        }
}

class BasketOfApples extends Basket {
        void put(Fruit fruit) {
                if(fruit instanceof Apple)
                        System.out.println("put a fruit into a basket of apples")
        }
}

class BasketOfOranges extends Basket {
        void put(Fruit fruit) {
                if(fruit instanceof Orange)
                        System.out.println("put a fruit into a basket of oranges")
        }
}


public class Sample {
```

```
        public static void putFruit(Basket basket, Fruit aFruit) {
                System.out.println(aFruit)
        }

        public static void main(String[] args) {
                putFruit(new BasketOfApples(), new Apple());
                putFruit(new BasketOfOranges(), new Orange)
        }
}
```

Now we can use similar code to do the exact same thing but this time using abstract types in Scala.

```
class Fruit { }
class Apple extends Fruit { }
class Orange extends Fruit { }

abstract class Basket {
        def put(fruit : Fruit) {
                println("put a fruit into a basket")
        }
}

class BasketOfApples extends Basket {
        type Item = Apple
        override def put(fruit: Apple) {
                println("put a Apple into a basket of apples")
        }
}

def putAFruit(basket : Basket, fruit : Fruit) {
        basket.put(fruit)
}

val basketOfApples = new BasketOfApple
val anApple = new Apple
basketOfApples.put(anApple)
putAFruit(basketOfApples, anApple)
```

How about we assign an `Animal` to item in another function, what happens there?

```
class Fruit { }
class Apple extends Fruit { }
class Orange extends Fruit { }
```

```
abstract class Basket {
        def put(fruit : Fruit) {
                println("put a fruit into a basket")
        }
}

class BasketOfApples extends Basket {
        type Item = Apple
        override def put(fruit: Apple) {
                println("put a Apple into a basket of apples")
        }
}

class BasketOfOranges extends Basket {
        type Item = Animal
        override def put(fruit: Apple) {
                println("put a Apple into a basket of apples")
        }
}


def putAFruit(basket : Basket, fruit : Fruit) {
        basket.put(fruit)
}

val basketOfApples = new BasketOfApple
val anApple = new Apple
basketOfApples.put(anApple)
putAFruit(basketOfApples, anApple)
```

Scala accepts the type of our abstract type `item`. But `Animal` would end up breaking our code, because we don't want a basket of `Animal` objects here we want a basket of `Orange` objects.

```
class Fruit { }
class Apple extends Fruit { }
class Orange extends Fruit { }

abstract class Basket {
        type Item <: Fruit
        def put(fruit : Item) {
                println("put a fruit into a basket")
        }
}
```

```
class BasketOfApples extends Basket {
        type Item = Apple
        override def put(fruit: Apple) {
                println("put a Apple into a basket of apples")
        }
}

class BasketOfOranges extends Basket {
        type Item = Animal
        override def put(fruit: Apple) {
                println("put a Apple into a basket of apples")
        }
}

def putAFruit(basket : Basket, fruit : Fruit) {
        basket.put(fruit)
}

val basketOfApples = new BasketOfApple
val anApple = new Apple
basketOfApples.put(anApple)
putAFruit(basketOfApples, anApple)
```

Here our code will break because `Animal` is not derived from `Fruit` because our parameter in the `put` method is restricted to either a `Fruit` or something that derives from a `Fruit`.

- `Item <: Fruit` means that we are restriction to `Fruit` and its child classes.

```
class Fruit { }
class Apple extends Fruit { }
class Orange extends Fruit { }

abstract class Basket {
        type Item <: Fruit
        def put(fruit : Item) {
                println("put a fruit into a basket")
        }
}

class BasketOfApples extends Basket {
        type Item = Apple
        override def put(fruit: Apple) {
                println("put a Apple into a basket of apples")
        }
}
```

```
class BasketOfOranges extends Basket {
        type Item = Orange
        override def put(fruit: Orange) {
                println("put a Apple into a basket of apples")
        }
}

def putAFruit(basket : Basket, fruit : Fruit) {
        basket.put(fruit)
}

val basketOfApples = new BasketOfApple
val anApple = new Apple
basketOfApples.put(anApple)
putAFruit(basketOfApples, anApple)
val basketOfOranges = new BasketOfOranges
val anOrange = new Orange
basketOfOranges.put(anOrange)
putAFruit(basketOfOranges, anOrange)
```

When you define an abstract type, it is not a standalone class. This `Item` is a class that is not known on the outside. It's encapsulated. It's existence is only known in the instance of an object.

So how do you really know what you're dealing with?

```
class Fruit { }
class Apple extends Fruit { }
class Orange extends Fruit { }

abstract class Basket {
        type Item <: Fruit
        def put(fruit : Item) {
                println("put a fruit into a basket")
        }

        def aFruitYouSupport : Item
}

class BasketOfApples extends Basket {
        type Item = Apple
        override def put(fruit: Apple) {
                println("put a Apple into a basket of apples")
        }
```

```
        override def aFruitYouSupport : Item = new Apple

}

class BasketOfOranges extends Basket {
        type Item = Orange
        override def put(fruit: Orange) {
                println("put a Apple into a basket of apples")
        }

        override def aFruitYouSupport : Item = new Orange

}

def putAFruit(basket : Basket, fruit : Fruit) {
        basket.put(basket.aFruitYouSupport)
}

val basketOfApples = new BasketOfApple
val anApple = new Apple
basketOfApples.put(anApple)
putAFruit(basketOfApples)
val basketOfOranges = new BasketOfOranges
val anOrange = new Orange
basketOfOranges.put(anOrange)
putAFruit(basketOfOranges)
```

Notice what we did, in `putAFruit` we put a fruit into the basket, but our fruit will be **polymorphic**, meaning we ask for it from the `basket` object itself.

We said a `Basket` will accept an `Item` but the `Item` is an abstract type. Which has a restriction that it has to be either a `Fruit` or derived from a `Fruit`.

- Then we specialized the `BasketOfApple` and `BasketOfOranges` from the `Basket`. And in this method we gave our `Item` a concrete type.

We used an abstract type in the parent class and a concrete type in the child classes.

- in Scala, we are able to override and call a method from the base class in a derived class. We are able to receive an `Item` as our concrete type `Apple` in `BasketOFApples` and we're also able to send an `Item` to the parent class so long as it fit our restrictions defined in the abstract type.