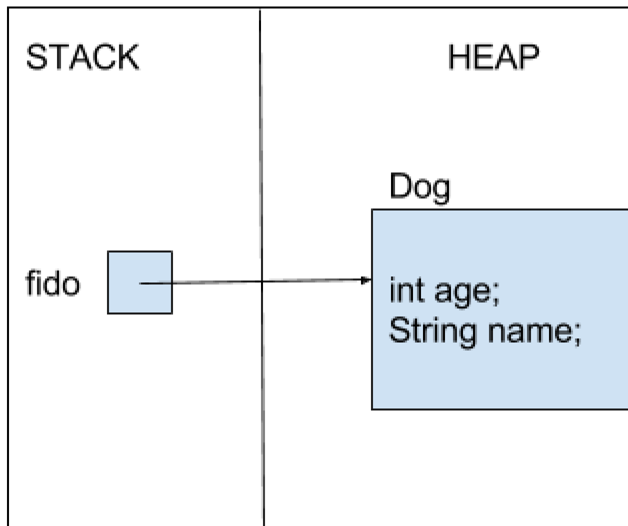
STATIC/DYNAMIC TYPE AND BINDING

Definition of Types

Static type refers to the type that an object is *declared* as, while dynamic type refers to the type that is *instantiated*:

```
Animal fido = new Dog();
```



Above, we see that fido is declared as an `Animal` and initialized as a `Dog`. Thus, its static type is `Animal`, and its dynamic type is `Dog`. If you prefer box and pointer diagrams, you can think of the static type as the box that is set aside for fido in the Stack. That box can hold the address for anything that is an `Animal`. The dynamic type is the type of the object at the address that the box *actually* points to. In this case, it is of type `Dog`.

Binding: Compile vs Run-time

Now that we know what static/dynamic type is, we start on the concept of static/dynamic binding. Binding refers to the link between method call and method definition. In other words, when a method is called, how do we know which method definition from which class to actually use?

For the most part, during compile time, Java will use the static type of the object to see if a method or variable exists. During run time, Java will use the dynamic type of the object to actually choose the method or variable to use. This is called *dynamic binding*.

There are a few exceptions to this. Variables, static methods, private methods, and final methods all use *static binding*. This means that during compile time, Java will use the static type of the object to choose the variable or method to use during run-time.

Since static binding happens during compile-time, it is sometimes referred to as early binding. Since dynamic binding happens during run-time, it is sometimes referred to as late binding.

Here are a few classic examples of these concepts below:

```
class Animal {
    public void swim(){System.out.println("Animal swim");}

    public static wave() {System.out.println("Animal wave");}
}

class Dog{
    public void swim() {System.out.println("Dog swim");}

    public static wave() {System.out.println("Dog wave");}
}
```

```
Somewhere else...
    public static void main(String... args) {
        Animal fido = new Dog();
        fido.swim();        //Prints "Dog swim"
        fido.wave();        //Prints "Animal wave"
    }
```

Explanation:

- `fido.swim()`:
  During compile time, Java checks that the `Animal` class (the static type) does indeed have the method `swim` with no arguments. During run time, Java looks for a method swim with no arguments from the dynamic type's class, `Dog`. Thus, the `Dog` class' swim method is called.
- `fido.wave()`:
  During compile time, Java checks that the `Animal` class (the static type) does indeed have the method wave with no arguments. However, you'll notice that wave is a static method, and thus an exception to dynamic-binding. Since it is one of the few types of methods that are statically bound, this means the method is chosen in compile-time, based on the static type of the object. So, Java goes ahead and binds the `fido.wave()` method call to the `Animal` class' no-argument wave method during compile-time. Thus, when we later run the main method, we already know to call the `Animal` class' no-argument wave method.

Casting

Casting allows us to change the static type of an object for **one line only**. Since this changes static type, its effects are only in compile-time (although as you'll see later, compile-time effects may influence run-time).

You are allowed to cast any object into something within its family. As an example, you can cast an `Animal` to a `Dog` (downcasting), or cast a `Dog` to a `Animal` (upcasting). Upcasting is casting to a supertype, while downcasting is casting to a subtype. Upcasting is always allowed, but downcasting can throw a ClassCastException if you do an incorrect cast (for example if something is of dynamic type `Animal` and you've cast it to a `Dog`, you will get this exception).

You may find casting to be useful if, for whatever reason, some object you're working with comes in with a static type that is not specific enough. There are, of course, a few dangers to this:

```
class Animal {
    public void swim(){System.out.println("Animal swim");}

    public static wave() {System.out.println("Animal wave");}
}

class Dog{
    public void swim() {System.out.println("Dog swim");}

    public void bark() {System.out.println("Woof");}
}
```

```
Version 1:
    public static void main(String... args) {
        Animal fido = new Dog();
        fido.bark();        //Gives a compile-time error
    }
```

```
Version 2:
    public static void main(String... args) {
        Animal fido = new Dog();
        ((Dog) fido).bark();     //Prints "Woof"

        Animal beast = new Animal();
        ((Dog) beast).bark();    //RunTimeException: ClassCastException
    }
```

- `fido.bark():`
  As you can see, in Version 1, we get a compile-time error. This is because fido is of static type Animal. So, in compile-time, Java will look for a no-argument method called bark in the Animal class. This does not exist, so a compile-time error occurs.

- `((Dog) fido).bark():`
  In Version 2, we avoid this by casting fido into a Dog. So, in compile time, fido's static type is temporarily changed to Dog for that one line. Thus, we look in the Dog class for a no-argument method called bark, and successfully find such a method. Thus, it passes compile-time. During run-time, Java uses the Dog dynamic type of fido to call bark(). Thus, "Woof" is printed out.

- `((Dog) beast).bark():`
  In the final example, we make a beast that is of static and dynamic type Animal. Since we cast beast to a Dog, its static type is changed to Dog for that one line. Thus, during compile-time Java uses the Dog static type of beast, confirms that there is a no-argument method called bark within the Dog class, and allows this to pass the compile-time check as well. However, during run time, you'll get a ClassCastException because the actual, dynamic type of beast is Animal. Thus, casting it to a Dog is an error.

Overloaded and Overriden Methods (Definition)

Be aware the overloaded methods are statically bound in compile-time, while overriden methods are dynamically bound in run-time.

In the example above (with `Animal` and `Dog`), `swim` and `wave` are both overriden methods in the `Dog` class.

There's one more tricky thing that happens with overloaded and overriden methods. Thus far, we haven't talked about methods that are both overloaded and overriden. Examine the code on the next page:

```java
public class Parent {
   public void greet(Parent p) {
       System.out.println("Parent greet Parent");
   }

   public void greet(Child c) {
       System.out.println("Parent greet Child");
   }
}

public class Child extends Parent {
   public void greet(Parent p) {
       System.out.println("Child greet Parent");
   }

   public void greet(Child c) {
       System.out.println("Child greet Child");
   }
}
```

Notice how `greet` is overloaded (it can take in a `Child` or a `Parent`), and overriden (the `Child` class redefines the method). Pay attention to what we call the "method signature": the method name and the number/type of the arguments it takes in. In this case, we have two method signatures: `greet(Parent p)`, and `greet(Child c)`

Okay, so let's run the main method below:

```java
public static void main(String[] args) {
   Parent c1 = new Child();
   c1.greet(c1);             //Prints "Child greet Parent"
   c1.greet((Child) c1);     //Prints "Child greet Child"

   Child c2 = new Child();
   c2.greet(c1);             //Prints "Child greet Parent"
}
```

Does the output of these methods surprise you?

This is because, during compile time, Java does not just check if the static type class has the method in question. Java also remembers the method signature that is found during compile time.

- Let's examine the case of `c1.greet(c1):`

During compile time, Java will check that the static class, `Parent`, does indeed have a greet method that can take in a `Parent` type (remember, `c1`'s static type is `Parent`, so we are looking for an argument that can take in a `Parent`). We successfully find a `greet` method that takes in a `Parent` as an argument. We then save this method signature for later. During run-time, Java will use the dynamic type of `c1`, `Child`, to find a method with the same method signature that we saved before. Thus, within the `Child` class we look for a `greet` method that can take in a `Parent` type. This is why "Child greet Parent" is printed out.

- Next, we look at `c1.greet((Child) c1):`

We cast the argument of `greet` to have the static type of `Child` for that line only. So, we look at the static class of `c1`, `Parent`, to find a `greet` method that takes in a `Child` type. We successfully find a `greet` method that takes in a `Child` as an argument. We then save this method signature for later. During run-time, Java uses the dynamic type of `c1`, `Child`, to find a method with the same method signature that we saved for this line. Thus, within the `Child` class we look for a `greet` method that can take in a `Child` type. This is why "Child greet Child" is printed out with the cast.

- Finally, let's examine `c2.greet(c1):`

During compile time, Java will check that the static class, `Child`, does indeed have a `greet` method that can take in a `Parent` type (again, `c1`'s static type is `Parent`, so we are looking for an argument that can take in a `Parent`). We successfully find a `greet` method that takes in a `Parent` as an argument. We then save this method signature for later. During run-time, Java will use the dynamic type of `c1`, `Child`, to find a method with the same method signature that we saved before. Thus, within the `Child` class we look for a `greet` method that can take in a `Parent` type. This is why "Child greet Parent" is printed out.