

## Week 10 - Polymorphism

---

### Description

It's time to dive into Polymorphism, the third and final pillar of object-oriented programming (the other two are Encapsulation and Inheritance), and how it can make our programs more flexible.

### Polymorphism

#### What is polymorphism?

The term polymorphism means "having many forms". In object-oriented programming, *polymorphism allows you assign a subclass object to a superclass variable*. So:

- A variable can refer not only to an object of its declared class, but also to an object of any class that is related to its declared class by inheritance.
- A polymorphic reference variable can refer to different types of objects over time.

That's it in a nutshell, but it's more powerful than just that. Not only can you assign a subclass object to a superclass variable:

- When you call a method using the superclass variable, the **actual** method executed will change from one execution to the next based on what type of object is **actually** assigned to the superclass variable.

You can also use polymorphism when passing objects into methods:

- You can code a method with a superclass type as a parameter, and actually call it using a subtype object instead.

We'll explore all these things in more detail below.

First, let's look at an example.

#### Assign a Subclass Object to a Superclass Variable

In the BankAccount class from previous weeks, we had a superclass called BankAccount, and subclasses called SavingsAccount and ChequingAccount.

ChequingAccount and SavingsAccount extend the behavior of BankAccount. This is the inheritance part you already know about.

Here comes the polymorphism part. ChequingAccount and SavingsAccount objects can be used by any code designed to work with BankAccount variables. If a method expects a parameter of type BankAccount, you can hand it a ChequingAccount object and it just works. All the BankAccount code can be used by anyone with a ChequingAccount in hand. This feature is known as polymorphism. A single object like ChequingAccount can have many (poly-) forms (-morph) and can be used as both a ChequingAccount object and a BankAccount object.

In the test program, we could do the following:

```
BankAccount other = new SavingsAccount(10000);
```

## Week 10 - Polymorphism

---

You might think this is impossible, since we have different class types on either side of the assignment operator: “other” is a BankAccount, but it is being assigned a SavingsAccount. This is **not** an error because SavingsAccount is a subclass of BankAccount. **You can assign a subclass object to a superclass variable (but not vice versa).**

Let's take this to a more detailed explanation:

```
BankAccount myAccount;  
  
//the next statement is nothing new, and is valid  
myAccount = new BankAccount(1000000);
```

The myAccount variable may be used to point to an object created by instantiating the BankAccount class, as above. However, it doesn't have to. The variable type (BankAccount) and the object it refers to must be compatible, but their types need not be exactly the same. The relationship between a reference variable and the object it refers to is more flexible than that. The variable can refer not only to an object of it's own class, but also to an object of any class that is related to it by inheritance. This means that, if ChequingAccount is related to BankAccount by inheritance, myAccount can refer to an object of the ChequingAccount class:

```
BankAccount myAccount;  
  
//this one is valid also, but here we assign  
//a ChequingAccount object to the myAccount variable  
myAccount = new ChequingAccount(1000000);
```

Remember that inheritance establishes an "IS-A" relationship, and a ChequingAccount **is a** BankAccount, therefore assigning a ChequingAccount object to a BankAccount reference is perfectly logical.

Note that the variable myAccount can refer to different types of objects at different times, so it can start out as a BankAccount object and later be changed to a ChequingAccount object.

### Polymorphic Method Calls

With polymorphism, objects of **different** types can call methods of the **same** type. For example, given our superclass BankAccount, the programmer can **override** BankAccount's deposit method in a subclass such as ChequingAccount. No matter what myAccount has **actually** been assigned to our myAccount variable, calling the deposit method for it will return the correct results by executing the deposit method for the correct, **matching** class. So, in the statement:

```
myAccount.deposit(10000.0);
```

if myAccount has been assigned a BankAccount object as in the first assignment statement above, it will execute the deposit method in the BankAccount class. However, if it has been assigned a ChequingAccount object, as in the second

## Week 10 - Polymorphism

assignment statement above, it will execute the deposit method in the ChequingAccount class.

So, if this line of code is in a loop used to process an array of objects that can be BankAccounts or ChequingAccounts, or if it's in a method that's called more than once, that line of code could call a different version of the deposit method each time it is called. For example, if it's called in the following loop:

```
for(BankAccount b : accounts) {  
    b.deposit(10000.0);  
}
```

and the accounts array contains both BankAccounts and ChequingAccounts, and perhaps SavingsAccounts, the deposit method of the BankAccount class will be called when myAccount references a BankAccount or SavingsAccount (since SavingsAccount does not override BankAccount's deposit method), and the deposit method of the ChequingAccount class will be called when myAccount references a ChequingAccount. **The type of the object, not the type of the reference, determines which version of a method is invoked.**

Here's another example:

**OK, OK maybe an example will help.**

```
Animal[] animals = new Animal[5];
```

```
animals [0] = new Dog();
```

```
animals [1] = new Cat();
```

```
animals [2] = new Wolf();
```

```
animals [3] = new Hippo();
```

```
animals [4] = new Lion();
```

```
for (int i = 0; i < animals.length; i++) {
```

```
    animals[i].eat();
```

```
    animals[i].roam();
```

```
}
```

Declare an array of type Animal. In other words, an array that will hold objects of type Animal.

But look what you get to do... you can put ANY subclass of Animal in the Animal array!

And here's the best polymorphic part (the *raison d'être* for the whole example), you get to loop through the array and call one of the Animal-class methods, and every object does the right thing!

When 'i' is 0, a Dog is at index 0 in the array, so you get the Dog's eat() method. When 'i' is 1, you get the Cat's eat() method

Same with roam().

At some point, the commitment is made to execute certain code to carry out a method call. This commitment is called **binding** a method call to a method definition. In many situations, the binding of a method usually happens at compile time. However, with

## Week 10 - Polymorphism

polymorphic references, the decision cannot be made until run time, because the compiler doesn't yet know which class the variable will refer to. The binding is done at run time at the moment the call is made - this is called **late binding** or **dynamic binding**. It is slightly less efficient than regular binding, but the overhead is often acceptable as a tradeoff for the flexibility a polymorphic reference provides.

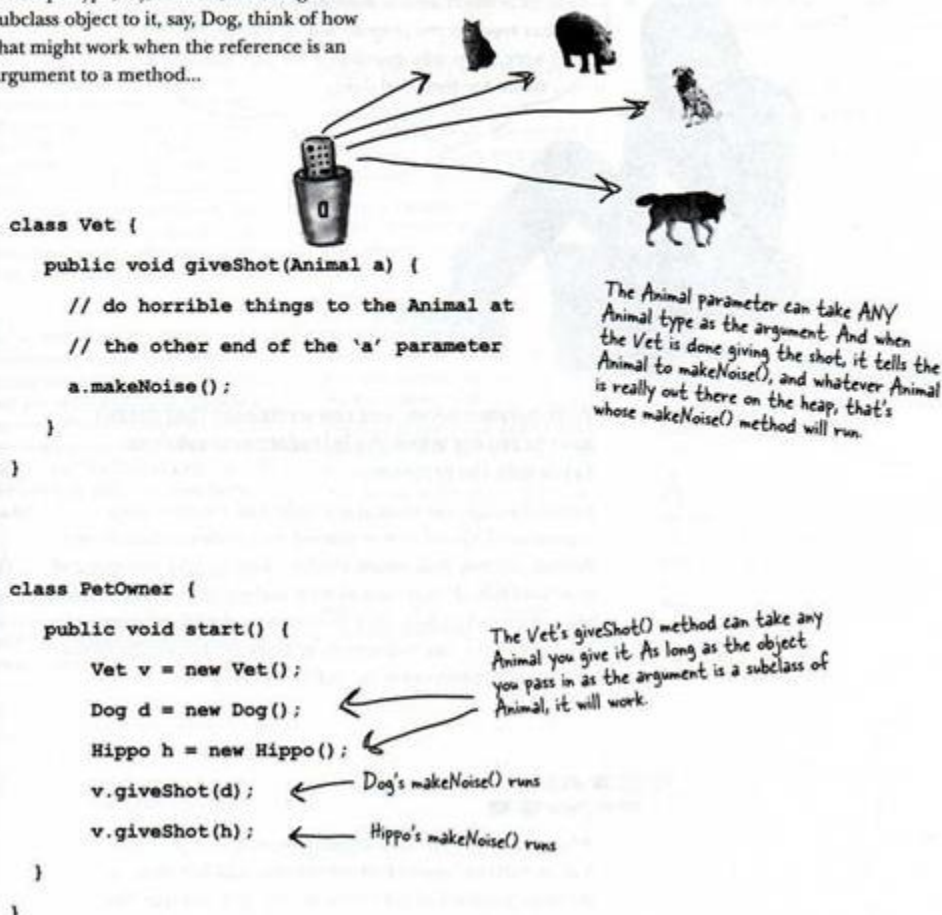
### Polymorphic Parameters and Return Types

A very common use of polymorphism is when you call methods with parameters whose data type is a class. In the following, assume that Dog and Hippo are subclasses of Animal:

#### But wait! There's more!

##### You can have polymorphic arguments and return types.

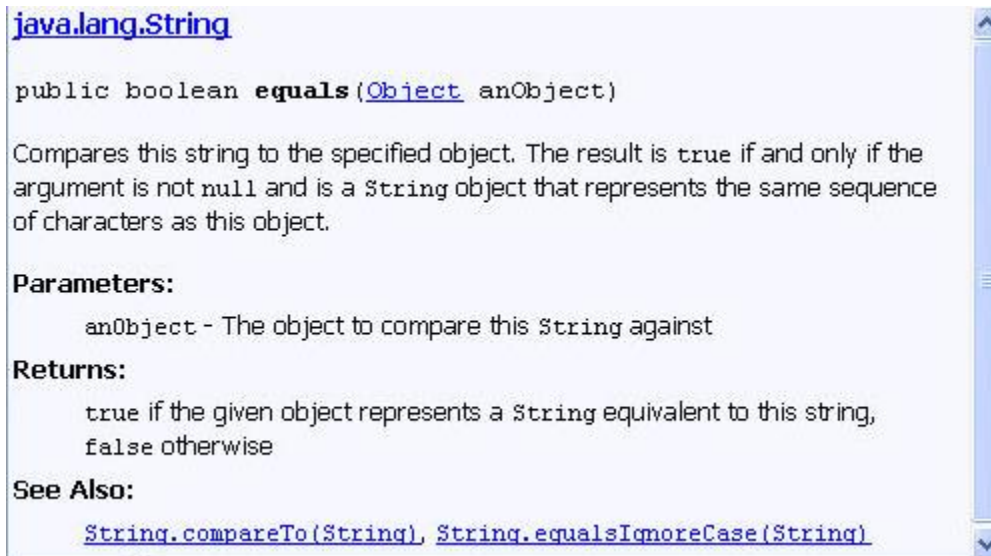
If you can declare a reference variable of a supertype, say, Animal, and assign a subclass object to it, say, Dog, think of how that might work when the reference is an argument to a method...



If you want to make your arguments very generic, you can make them Objects, but there are some drawbacks to doing this. Some of the Java API methods have Object

## Week 10 - Polymorphism

arguments to allow maximum flexibility. For example, in the `String` class, the `equals` method:



[java.lang.String](#)

```
public boolean equals(Object anObject)
```

Compares this string to the specified object. The result is true if and only if the argument is not null and is a `String` object that represents the same sequence of characters as this object.

**Parameters:**

- `anObject` - The object to compare this `String` against

**Returns:**

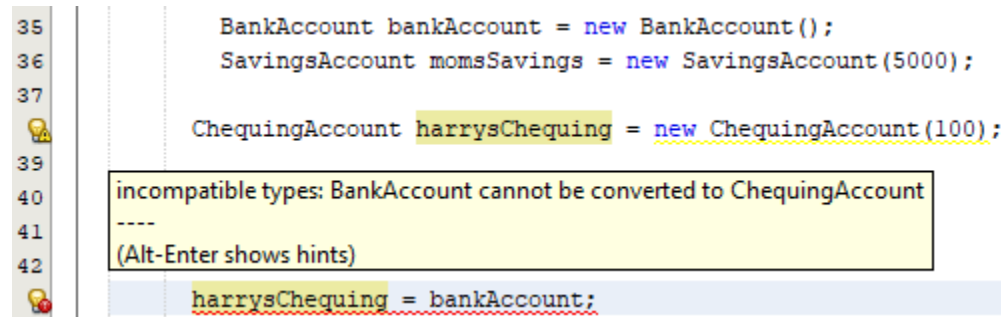
- true if the given object represents a `String` equivalent to this string, false otherwise

**See Also:**

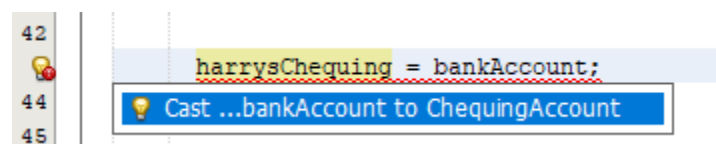
- [String.compareTo\(String\)](#), [String.equalsIgnoreCase\(String\)](#)

### Casting Objects and instanceof operator

We can declare a `BankAccount` reference variable and assign a `ChequingAccount` object to it. You can always assign subclass objects to superclass variables. In contrast, you **cannot** assign superclass objects to subclass variables. You will get a compile-time error if you try to:



```
35      BankAccount bankAccount = new BankAccount();
36      SavingsAccount momsSavings = new SavingsAccount(5000);
37
38      ChequingAccount harrysChequing = new ChequingAccount(100);
39
40      incompatible types: BankAccount cannot be converted to ChequingAccount
41      ----
42      (Alt-Enter shows hints)
43
44      harrysChequing = bankAccount;
```



```
42
43
44      harrysChequing = bankAccount;
45
```

Cast ...bankAccount to ChequingAccount

If you want to test to see if an object is an instance of a particular class, you can use the **instanceof** operator:

```
if (account instanceof BankAccount) {
    ...}
```

## Week 10 - Polymorphism

---

Note that instanceof is an operator, not a method, so it is all lowercase and has no parentheses. The expression above returns a boolean - true if account is a BankAccount and false if it is not.

You need to cast if you want to use an object that contains a reference to the superClass to call a method that only exists in the subclass. In the code below, the accounts array contains a combination of BankAccount, SavingsAccount and ChequingAccount objects. The SavingsAccounts need interest added, the ChequingAccounts need fees deducted, and the BankAccounts don't need anything done to them. So, in the loop, we use instanceof to find the type of the current object. If it is a SavingsAccount we call the addInterest method. If it is a ChequingAccount, we call the deductFees method:

```
for (BankAccount account: accounts) {
    if (account != null) {
        //is the object actually a savings account?
        //if so, add interest
        if (account instanceof SavingsAccount){
            //call methods that are only in
            //SavingsAccount class - need to cast to do
            //it - note the brackets around the
            //cast especially the one to the
            //right of account!!
            ((SavingsAccount)account).addInterest();
        } //if it's a chequing account, deduct fees
        else if (account instanceof ChequingAccount){
            ((ChequingAccount)account).deductFees();
        }
    }
}
```

Note that we have to specifically cast the object to a SavingsAccount or ChequingAccount before we can call a method that exists only in the SavingsAccount or ChequingAccount class (notice the brackets around the cast, especially to the right of account). Casting, as used above, does not change the data type of the object being cast. It just tells the compiler that you know the object being acted on really belongs to a subclass.