Object Interaction in Java – Polymorphism & `abstract` Modifier

Part 1: Polymorphism

Recall the Person and Student classes, which might have the following definitions (the dots, `...` indicate there may be more to each definition).

```
class Person
{
 ...
}

class Student extends Person
{
 ...
}
```

This code has the obvious hierarchy of: `Person --> Student`
More correct, but less obvious, is the hierarchy: `Object --> Person -->`

`Student` Consider the following code:

```
Person p1, p2;
p1 = new Person(...);
p2 = new Student(...);

System.out.println(p1.toString());
System.out.println(p2.toString());
```

In both cases, `p1` and `p2` are references to the data type `Person`. At program execution time, however, they refer to different types of objects because of the way they were constructed. As a result, each call to `toString()` is actually calling a different method, one from the `Person` class, one from the `Student` class.

This does not actually cause a problem with Java, where the appropriate method to use is automatically decided upon at runtime. A situation like this, where the method invoked depends on the associated object type, rather than the defined object type, is called **polymorphism**.

Object Interaction in Java – Polymorphism & `abstract` Modifier

Part 2: Abstract Class

To help explain and illustrate the concept of an **abstract class**, we will consider an example of class definitions designed to handle bank account information. We might define a general class called Account which contains fields and methods required by all accounts. We could then define classes Savings and Chequing which both extend Account. Each of these new subclasses would contain only the additional fields and methods which are unique to either savings or chequing accounts.

For example, if all types of accounts should give interest, but the rates differ between account types, then both accounts might have their own interestRate field. Taking this further, the rules for calculating interest may actually vary between accounts (e.g., depending on minimum balances, length of time that money has been in the account, etc.). In such a case, unique methods may also be required to computeInterest().

At some point in the future, we may wish to add other account types, and they will require many custom definitions, just as the savings and chequing accounts did.

It is possible to create an abstract superclass which forces all subclasses to define their own implementations of any methods marked as abstract.

```
abstract class Account
{
 double interestRate;
 abstract double computeInterest (int accountNumber);
}

class Savings extends Account
{
 double computeInterest (int accountNumber)
 {
 // implementation here
 }
}
```

```
class Chequing extends Account
{
 double computeInterest (int accountNumber)
 {
 // implementation here
 }
}
```