Week 9 Description

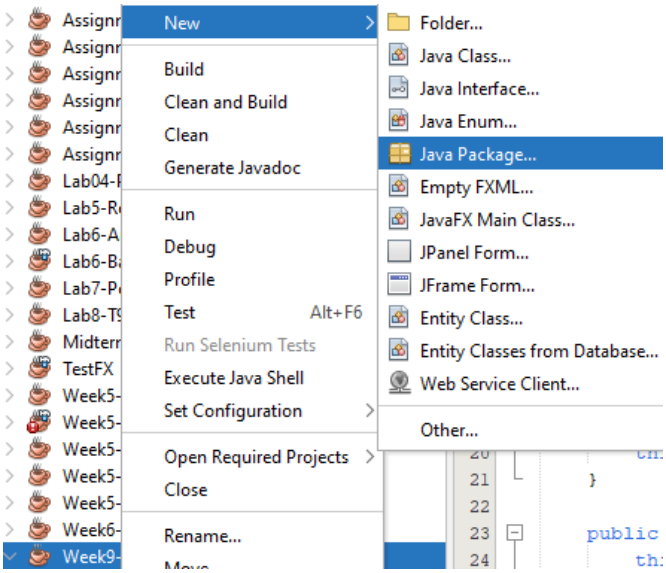
This week, we will implement inheritance in our algorithms.

We'll start, however, by discussing the idea of creating our own packages to make reuse easier.

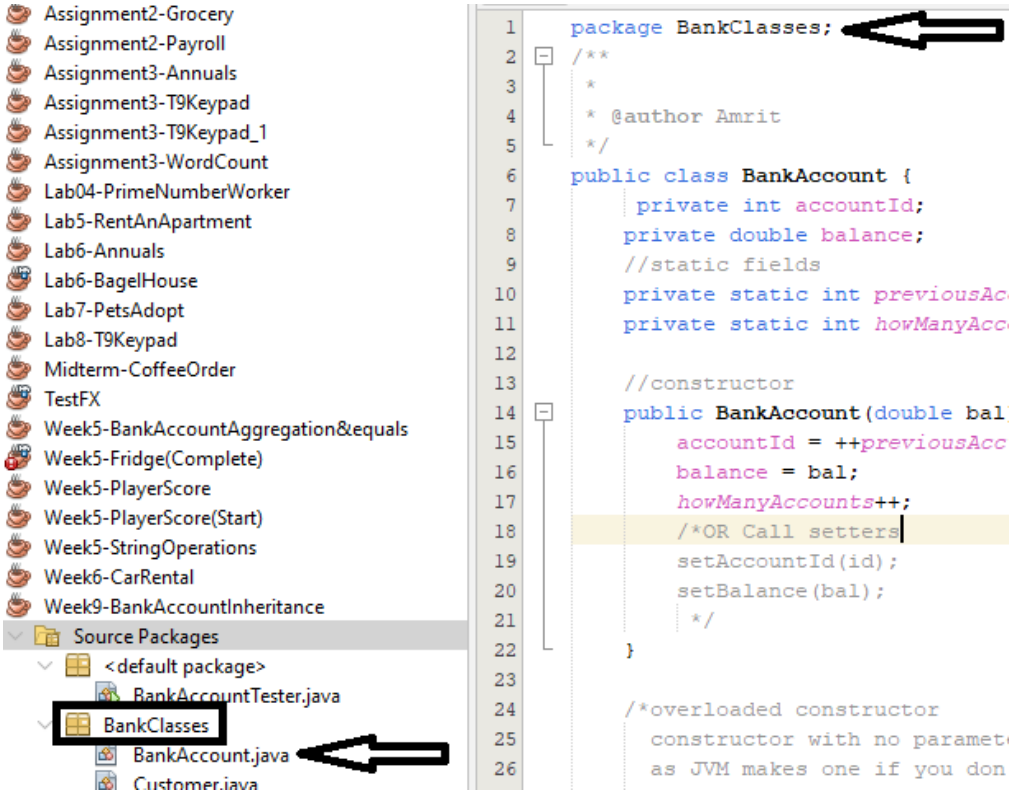
Reusing objects – Packages

One of the big benefits of Object Oriented programming and creating our own classes is that we can reuse them in other projects and avoid reinventing the wheel (not to mention retesting the wheel!). To reuse one class, we can simply copy and paste it into another project. To reuse several related classes, it's best to put the classes into a package, and use the package in other projects.

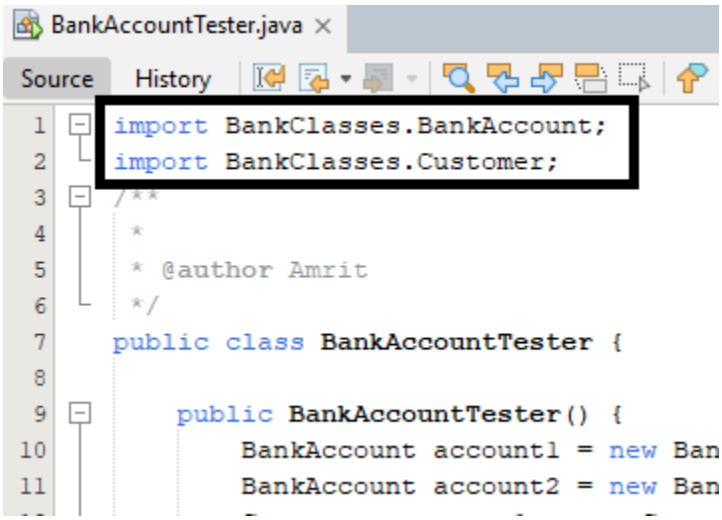
To add a package to your project, in the Package Explorer window, right-click source package and choose New, Package. Name your package whatever you like, and click Finish.



You can drag and drop a class from one package to another. In the screen shot below, I've dragged BankAccount.java from the default package to the new package I created called BankClasses. If you look at the class, you'll see that the class includes, as the very first statement, a package statement that indicates that the BankAccount class is in a package called BankClasses. NetBeans has automatically added this package statement to the BankAccount class:



Notice on the left in the package explorer that the project now includes the package name. To use a class from another package, we import it just as we would a Java class:



To add a new class to a package, just create the class as usual by selecting the package and choosing New, Class.

Inheritance

Inheritance *allows an object of one class to acquire the properties and methods of another class*.

Inheritance is central to object-oriented (OO) programming in Java and other OO languages. The key concept is creating a new class from an existing one. The new class is called a **subclass** and the original one a **superclass**. Note the use of the Java keyword **extends** in the header of the subclass definition. In the example below, GeometricObject is the superclass. Circle is the subclass:

```
public class Circle extends GeometricObject
```

The subclass includes all the members (both fields and methods) of the superclass and then can add additional members (fields and/or methods) or override the methods in the superclass:

For example, let's recall our simple bank account example from last week. BankAccount class pays interest, but interest belongs to Savings account

We will use inheritance to define a separate SavingsAccount class that does pay interest.

Defining a subclass

Now suppose that the bank wants to offer both the basic bank account, and also a savings account that has all of the behaviours of a bank account – it has a balance, and you can do deposits, withdrawals and transfers, and get your balance – but has the additional feature that it pays interest.

You could define a SavingsAccount class that was completely separate from the BankAccount class. You could copy and paste the code for the BankAccount class, then modify it to add new fields and methods for the savings account interest. This has some drawbacks, however. If you later modify the BankAccount class, you have to remember to copy the modifications to the SavingsAccount class. More importantly, if the definitions of BankAccount and SavingsAccount are completely separate, then we cannot write code that treats savings accounts as a special sort of bank account – we have to deal with objects of the two different classes separately.

Java (and other object-oriented languages) gives us a better way. In the header of the definition of SavingsAccount, we say that SavingsAccount **extends** BankAccount:

```
public class SavingsAccount extends BankAccount {
// body of class goes here
}
```

"extends" is a Java keyword. When one class extends another, we say that the new class is a **subclass** of the old one, and the original class is a **superclass** of the new one. In our example, SavingsAccount is the subclass and BankAccount is the superclass.

This means the SavingsAccount class automatically inherits all of the fields and methods of the BankAccount class. We do not have to write the declaration of the balance field, the deposit method, etc. They are already part of our SavingsAccount class. We only need to add whatever new fields, constructors and methods we want our new class to have.

The definition of the SavingsAccount class is given below:

```
public class SavingsAccount extends BankAccount {

  // Constructs a bank account with some balance.
  public SavingAccount(double balance){
       super(balance);
   }
  public void addInterest(){
      this.deposit(this.getBalance() * MyConstants.INTEREST_RATE);
  }
   //Method overloading
   public void addInterest(double newInterest) {
      this.deposit(this.getBalance() * newInterest);
   }
 }
```

Our SavingsAccount class has two fields: accountId and balance, inherited from BankAccount. It has eight methods: getAccountId, setBalance, getBalance, deposit, withdraw and transfer (inherited from BankAccount), and addInterest, addInterest(double) (newly defined in SavingsAccount).

Keyword "super"

Now, back to the BankAccount example. Here's how the addInterest method looks:

```
  public void addInterest() {
     double interest = getBalance() * MyConstants.INTEREST_RATE;
     deposit(interest);
  }
```

Notice how the addInterest method makes use of the getBalance and deposit methods. Could we have defined addInterest as follows?

```
public void addInterest() {
  double interest = balance * MyConstants.INTEREST_RATE;
  balance += interest;
}
```

Since a SavingsAccount object has a field called balance, this looks reasonable. It won't compile, however. Because balance was declared to be private in BankAccount,

we cannot directly reference it from outside that class, not even in a subclass like SavingsAccount. We can only work with it via public methods like getBalance and deposit. Similarly, in the constructor:

SavingsAccount(double initialBalance)

we might be tempted to write:

```
public SavingsAccount(double initialBalance) {
 balance = initialBalance;
}
```

This won't work because balance is private in the superclass. To set the initial balance, we need to call the constructor of the superclass BankAccount. We don't do this in the usual way, by writing new BankAccount(initialBalance).

This would compile, but the result would not be what we want: the constructor would create a new BankAccount object separate from the SavingsAccount that we're trying to construct. That's not what we want.

Instead, we call the BankAccount constructor by using the Java keyword "super", followed by an argument list in parentheses. This syntax is used inside a subclass constructor to call the constructor of its superclass. This call must be the first line in the body of the constructor. Calling super later in the subclass constructor, after other statements, is a syntax error.

Let's see the code for the constructors again:

```
  // Constructs a savings account with a given balance
  public SavingsAccount(double initialBalance) {
     super(initialBalance);
  }
```

You can see that the constructor for SavingsAccount has a call to super, while the no-argument constructor does not. In fact, **every** subclass constructor always calls its superclass constructor. If this call is not made explicitly as the first line in the body of the subclass constructor, then the compiler inserts a call to the no-argument constructor of the superclass. It is as if, for the no-argument constructor we had written:

```
public SavingsAccount() {
 super();
}
```

In this case, we can leave out the call to super, since the compiler will insert it for us. This will only work if the superclass HAS a no-argument constructor. If the superclass doesn't have such a constructor, you must explicitly call super in the subclass constructor.

In summary, a subclass inherits the fields and methods of the superclass that it extends. Constructors are not inherited, but a subclass constructor always calls a superclass constructor, either explicitly by using the keyword super with a parameter list, or implicitly (i.e. the compiler adds the call super(); as the first line of the body of the subclass constructor).

You can use the super notation to call any method from the super class, from any method in the subclass (it's not limited to constructors).

Protected Access

So far, we've used public and private access specifiers. There are actually two others: package and protected.

As mentioned briefly in an earlier lesson, if you don't specify an access type at all, the default is package, which means the field (property) is available to other classes in the same package. Specifying package does the same thing.

The last access specifier is protected, which, when used in a class definition, means that the class itself has access to it, and that all subclasses of the class also have access to it. This is useful when defining a hierarchy of classes if you want the subclasses to have complete access to a field, rather than having to use the get and set methods in the superclass to access it.

Now, back to the SavingsAccount example. We saw on the previous page that we CANNOT code the addInterest method like this:

```
public void addInterest() {
    double interest = balance * MyConstants.INTEREST_RATE;
    balance += interest;
}
```

Because balance is private in the BankAccount class, we can only work with balance via public methods like getBalance and setBalance. This works fine for most cases.

However, if we have a reason to, we can make use of another option, the **protected** access specifier. In BankAccount, we can make balance protected rather than private. This way, SavingsAccount can access this field directly, without using the getters and setters, and our program would work with either the first or the second version of addInterest shown above.

protected

Protected access is almost identical to package access, with one exception: it allows subclasses to *inherit* the protected thing, *even if those subclasses are outside the package of the superclass they extend.* That's it. That's *all* protected buys you—the ability to let your subclasses be outside your superclass package, yet still *inherit* pieces of the class, including methods and constructors.

Many developers find very little reason to use protected, but it is used in some designs, and some day you might find it to be exactly what you need. One of the interesting things about protected is that—unlike the other access levels—protected access applies only to *inheritance.* If a subclass-outside-the-package has a *reference* to an instance of the superclass (the superclass that has, say, a protected method), the subclass can't access the protected method using that superclass reference! The only way the subclass can access that method is by *inheriting* it. In other words, the subclass-outside-the-package doesn't have *access* to the protected method, it just *has* the method, through inheritance.

Overriding methods

There are basically three things that you can do in a subclass to make it different from the superclass it extends:

- Add new fields.
- Add new methods.
- Override inherited methods.

Now suppose our bank wants to offer another sort of bank account, a chequing account, that does not pay interest, but that levies a service charge to cover the cost of processing cheques. Customers are allowed a certain number of free transactions each month. If they exceed this number, the bank deducts a service charge from their account.

I have declared two more constants in MyConstants interface, changed class to interface.

```
private static final int FREE_TRANSACTIONS = 3;
private static final double TRANSACTION_FEE = 0.50;
```

Here's how we could implement chequing account:

```
/*
   A chequing account that charges transaction fees.
   The deposit and withdraw methods inherited from
   BankAccount are overridden
*/
public class ChequingAccount extends BankAccount {

   private int transactionCount;
```

```
  //   Constructs a checking account with a given balance
  public ChequingAccount(double initialBalance) {
    // construct superclass
    super(initialBalance);

    // initialize transaction count
    transactionCount = 0;
  }

  @Override
  public void deposit(double amount) {
    transactionCount++;
    // now add amount to balance
    super.deposit(amount);
  }

  @Override
  public void withdraw(double amount) {
    transactionCount++;
    // now subtract amount from balance
    super.withdraw(amount);
  }

  // Deducts the accumulated fees and resets the transaction count.
  public void deductFees() {
    if (transactionCount > MyConstants.FREE_TRANSACTIONS) {
      double fee = MyConstants.TRANSACTION_FEE *
        (transactionCount - MyConstants.FREE_TRANSACTIONS);
      super.withdraw(fee);
    }
    transactionCount = 0;
  }
}
```

In SavingsAccount, we did not have definitions for deposit and withdraw methods, but the class did have these methods, because they were inherited from the superclass, BankAccount. In ChequingAccount, we DO have definitions for deposit and withdrawal. Notice that these have the **same** signature (the same paramater list) in both BankAccount and ChequingAccount. This is **different from overloading**, where we have two or more methods with the same name but **different** parameter lists. When we have methods with identical signatures in a superclass and a subclass, we say that the definition in the subclass **overrides** the one inherited from the superclass.

Method overriding *allows a subclass to provide a different implementation of a method that is already provided by one of its superclasses. The subclass's method has the same name* ***and*** *parameter list as the superclass's overridden method.* The

implementation in the subclass overrides (replaces) the implementation in the superclass.

When we call the deposit method on a BankAccount object, the program uses the deposit method defined in BankAccount, in just the way we've seen previously. When we call the deposit method on a SavingsAccount object, the program uses the deposit method defined in BankAccount and inherited (unchanged) into SavingsAccount. When we call the deposit method on a ChequingAccount object, the program uses the deposit method defined in ChequingAccount, not the one from BankAccount. The new method definition has replaced the inherited one in ChequingAccount.

When you override an inherited method in a subclass, the method must have the same signature as in the superclass. It must also have the same return type. Trying to give it a different return type is a syntax error, and the program will not compile. The access specifier will usually be public in both the superclass method and the overriding subclass method. You are allowed to change the access specifier, but only in the direction of giving wider access, going in the direction

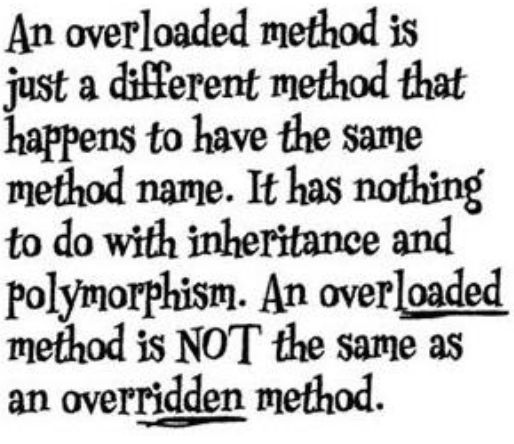
private -> package (no specifier) -> protected -> public.

You cannot change the access specifier in the opposite direction, to narrow the access to the method. If the method is private in the superclass, it can have any of the above four specifiers in the subclass. If it is protected in the superclass, it must be either protected or public in the subclass. If it is public in the superclass (the most common situation), it must also be public in the subclass.

Overriding vs. Overloading

When you are writing a subclass definition, you can override a method inherited from a superclass, or overload it, or both, or neither. By default, methods are inherited unchanged from a superclass to a subclass (e.g. deposit from BankAccount to SavingsAccount). If you write a method with the **same** name but a **different** signature in the subclass, then the inherited method still exists in the subclass along with the new method. You have two methods with the same name. This is **overloading**. Remember that ***overloaded methods*** *are methods with the same name but different parameters.*

If you write a method with the **same** signature in the subclass, then the new method replaces the inherited one, which no longer exists in the subclass. The new method has replaced it, and you no longer have the inherited method. This is **overriding**.



Let's go back to the definition of ChequingAccount. We needed to override the deposit method inherited from BankAccount, because we want to increment the count of how many transactions have been performed this month. We also need to perform the deposit, by adding the amount passed as an argument to the previous balance. You might think of implementing this by writing

```
public void deposit(double amount) {
  transactionCount++;
  // now add amount to balance
  balance += amount;
}
```

This won't work, because balance is private in BankAccount, and can't be referenced from outside that class.

You might then think of doing the following:

```
public void deposit(double amount) {
  transactionCount++;
  // now add amount to balance
  deposit(amount);
}
```

The problem with this is that the deposit method is calling itself. This new call to deposit will again call itself, and again, and again, and… This is called infinite recursion, and is similar to an infinite loop. It is not a syntax error, but is a logic error, since the program will never terminate.

Inside the deposit method of ChequingAccount, we don't want to call deposit as defined in ChequingAccount, but as defined in BankAccount. We do this by prefixing the method call with the keyword super:

```
public void deposit(double amount) {
  transactionCount++;
  // now add amount to balance
  super.deposit(amount);
}
```

This calls the deposit method as defined in the superclass.

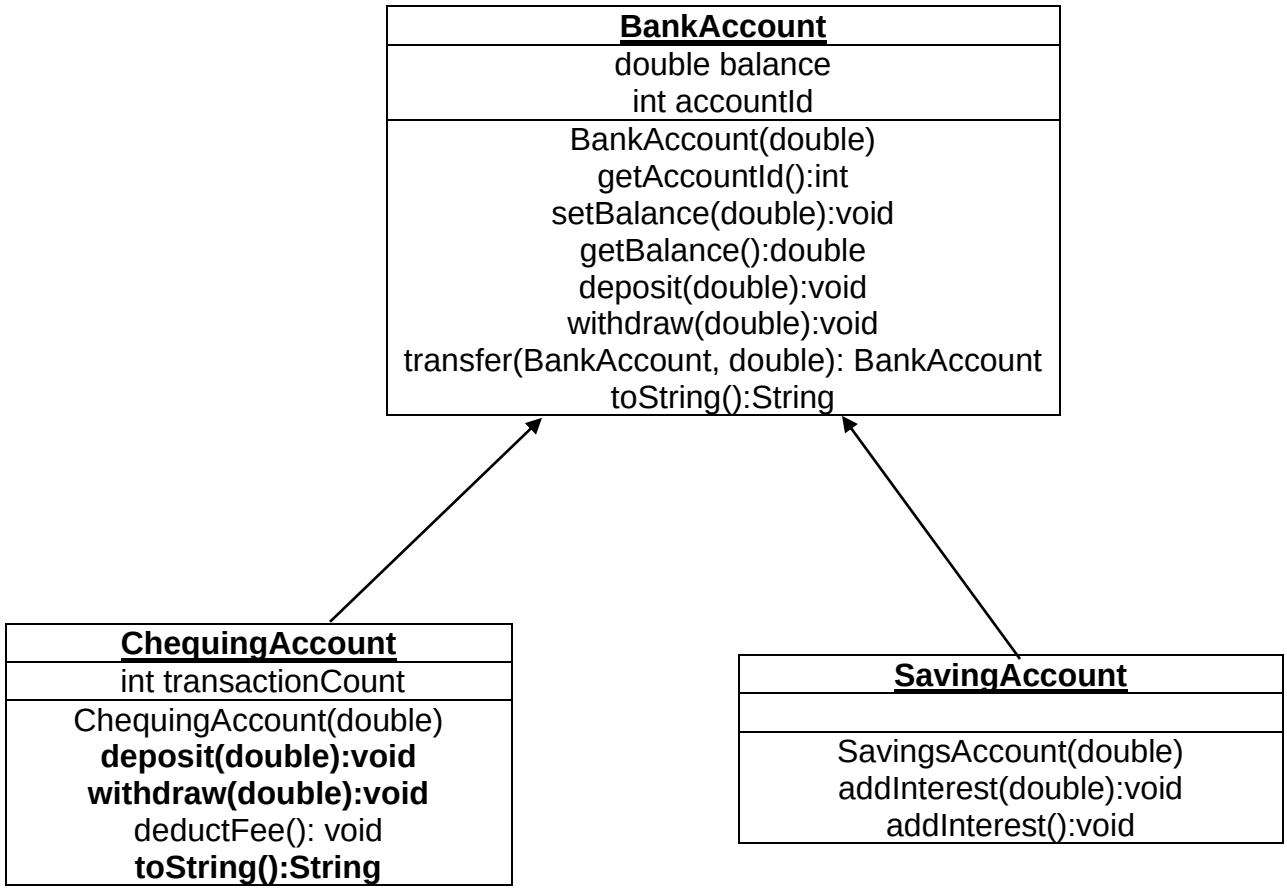
Here's a test program that makes use of SavingsAccount and ChequingAccount. Notice that the test class does not extend BankAccount, SavingsAccount, etc. It's just a class to hold a main method.

```
// This program tests the BankAccount class and its subclasses.
public class BankAccountTest {
   public static void main(String[] args) {
      SavingsAccount momsSavings = new SavingsAccount(5000);
```

```
    ChequingAccount harrysChequing = new ChequingAccount(100);

    harrysChequing.deposit(100);
    momsSavings.deposit(1000);
    harrysChequing.withdraw(80);
    momsSavings.transfer(harrysChequing, 2000);
    momsSavings.withdraw(1000);
    harrysChequing.transfer(momsSavings, 1000);

    // simulate end of month
    momsSavings.addInterest();
    harrysChequing.deductFees();

    System.out.println("Mom's savings balance = $" + momsSavings.getBalance());
    System.out.println("Harry's chequing balance = $"  + harrysChequing.getBalance());
  }
}
```

The IS-A Relationship

The UML representation of inheritance is an arrow with the arrowhead pointing up towards the superclass. Here's the UML diagram for the BankAccount hierarchy:



Due to inheritance, a SavingsAccount has all of the fields and methods of a BankAccount, so we can say that a SavingsAccount **IS A** BankAccount. Similarly, a ChequingAccount **IS A** BankAccount. Any instance of a subclass **IS AN** instance of its superclass. This is called the "IS A" relationship between subclasses and superclasses.

The opposite is not true. A BankAccount is not necessarily a SavingsAccount. Some bank accounts are savings accounts, but others are not.