Inheritance: Tyler Adam, Robert Lien, Charles McAnany.
Inheritance allows you to create new classes that incorporate the functions of other classes. For a superclass which contains **public**, **private**, and **protected** fields and methods, a subclass which inherits it will behave as though all of the superclass' **protected** and **public** members have been defined in the subclass' implementation. As an example, Appendix 1 shows a class, `BankAccount`. It contains fields `account_number` and `Balance`. It contains member functions `deposit`, `withdraw`, and `getBalance`. Should another class which behaves similarly to `BankAccount` be needed, one option is to write an entirely separate class `SavingsAccount` which behaves as would `BankAccount` except with further abilities. Appendix 2 shows this implemented. This, however, has several problems. First, it involves wide swaths of copied code, which creates unnecessarily large programs. Additionally, should the implementation of `BankAccount` be updated, any improvements to `BankAccount` will not cascade to `SavingsAccount`. A more robust method for creating specialized classes uses inheritance. For inheritance, one says that subclass `SavingsAccount2` **extends** `BankAccount`. (`SavingsAccount2` is presented in Appendix 3.) If no new functions are added to class `SavingsAccount2`, then an object of class `SavingsAccount2` will be identical to an object of class `BankAccount`. This would, of course, be silly. If new methods or fields are included in `SavingsAccount2`, then it will behave as would `BankAccount`, except that any calls to methods or fields defined in `SavingsAccount2` will result in `SavingsAccount2`'s code being executed.

Overriding a method is the defining of a method that is previously defined by the superclass. The code in appendix five demonstrates an overridden method. This newly created method must have the exact same signature as the one defined by the superclass but the implementation will be different. When an overridden method is called by an object, the object's class is searched for the method, but if the object's class does not contain the method, the immediate superclass is searched for the method. The searching through superclasses is continued until the most recently defined instance of the method is found. The declaration of "**super**." indicates the reference to the superclass' methods. Therefore, in order to call an overridden method belonging to a superclass, the declaration "**super**." must be included prior to the method name. In addition to overriding, a method can also be overloaded. An overloaded method is a method that contains the same method name as another method but takes in different signatures. Overloading of a method can occur within a single class or over inherited classes.

If a class contains a method stub, that is, a method that does not specify an implementation, then the class must be declared **abstract**. If a class contains no implemented methods, it may be declared an **interface**. (**interface**s are a subset of **abstract** classes.) **abstract** classes and **interface**s cannot be instantiated, that is, no objects can be created from them. However, they can be inherited to form classes which can be instantiated. These instantiable classes are known as "concrete." A (concrete) subclass which extends an **abstract** class must provide implementations for the method stubs. A class may only inherit one class which is not an **interface**. This prevents a class inheriting two implemented functions with identical signatures, which would lead to ambiguity. A class may inherit multiple **interface**s, since the methods are not implemented and that prevents ambiguity.

In general, a subclass which inherits a superclass may be used in any code which expects an object of the superclass.
It is said that a class which inherits another class is an **instanceof** the superclass. This is a concept known as polymorphism, which is beyond the scope of inheritance.

Every class implicitly inherits at least one other class. If no superclass is specified in the code for a class, that class automatically inherits Object and its methods. Object contains several very general methods that are typically overridden in its subclasses.

One confusing part of inheritance occurs when a superclass method could refer to a subclass variable. Appendix 6 presents a case where a superclass contains a **private** variable which is modified by a **public** method. When a subclass creates another variable of the same name (which is **bad since it shadows the other**!), then calls the superclass method that accesses that variable, the superclass method changes the superclass' **private** field, not the subclass' field. When the subclass attempts to return the variable, it returns the subclass' instance of that variable. All of this confusion could have been avoided by using meaningful variable names.

Appendix 1. BankAccount class.

```
1  public class BankAccount extends Object{
2
3        private int account_number;
4        protected double balance;
5
6        public BankAccount(int account){
7              account_number = account;
8              balance = 0;
9        }
10
11       public void deposit(double value){
12             this.balance += value;
13       }
14
15       public void withdraw(double value){
16             if(this.balance < value){
17                   System.out.println("I can't let you do that, Starfox.");
18             }else{
19                   this.balance -= value;
20             }
21       }
22
23       public double getBalance(){
24             return this.balance;
25       }
26 }
```

Appendix 2. Savings account created without inheritance.

```
1  public class SavingsAccount {
2
3        private int account_number;
4        protected double balance;
5
6        public SavingsAccount(int account){
7              account_number = account;
8              balance = 0;
9        }
10
11       public void deposit(double value){
12             this.balance += value;
13       }
14
15       public void withdraw(double value){
16             if(this.balance < value){
17                   System.out.println("I can't let you do that, Starfox.");
18             }else{
19                   this.balance -= value;
20             }
21       }
22
23       public double getBalance(){
24             return this.balance;
25       }
26
27       public void addInterest(double rate){
28             balance += rate * balance;
29       }
30 }
```

Appendix 3. SavingsAccount created using inheritance.

```
1   public class SavingsAccount2 extends BankAccount{
2
3         private int account_number;
4
5         public SavingsAccount2(int account){
6               super(account);
7               account_number = account;
8         }
9
10        public void addInterest(double rate){
11              balance += rate * balance;
12        }
13  }
```

Appendix 4. Class to test the BankAccount classes.

```
1   public class InheritanceExample {
2
3         public static void main(String[] args) {
4
5               //Instantiate and interact with a simple BankAccount
6               BankAccount basic = new BankAccount(234235);
7               basic.deposit(130.45);
8               basic.withdraw(100.00);
9               System.out.println("Simple account balance: $" + basic.getBalance());
10
11              //Instantiate and interact with a SavingsAccount
12              SavingsAccount save = new SavingsAccount(234236);
13              save.deposit(130.45);
14              save.addInterest(.06);
15              save.withdraw(100.00);
16              System.out.println("\nSavings account balance: $" + save.getBalance());
17
18              //Instantiate and interact with a SavingsAccount
19              // that extends BankAccount
20              SavingsAccount2 save2 = new SavingsAccount2(234237);
21              save2.deposit(130.45);
22              save2.addInterest(.06);
23              save2.withdraw(100.00);
24              System.out.println("\nSavings 2 balance: $" + save2.getBalance());
25        }
26  }
```

Appendix 5. Overriding and overloading.

```
1   public class SuperClass {
2
3         public void displayMessage() {
4               System.out.println("displayMessage in SuperClass called.");
5         }
6
7         public void displayMessage(int number) {
8               System.out.println("SuperClass says: " + number);
9         }
10  }
```

```
1   public class SubClass extends SuperClass {
2
3         public void displayMessage(){
4               System.out.println("displayMessage in SubClass called.");
5         }
6
7         public void displayMessage(String arg){
8               System.out.println("SubClass says: "+arg);
9         }
10
11        public void displaySuperMessage(){
12              super.displayMessage();
13        }
14  }
```

```
1   public class OverridingTest {
2
3         public static void main(String[] args) {
4               SubClass C = new SubClass();
5               C.displayMessage(5);
6               C.displayMessage();
7               C.displaySuperMessage();
8               C.displayMessage("Hello!");
9         }
10  }
```

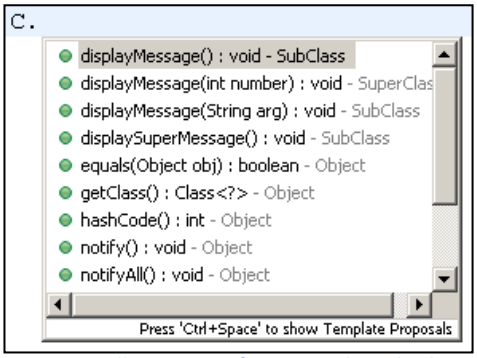

Figure 1. Illustration of suggested methods. Each method lists the class from which it is derived.

Console output:

```
BaseClass says: 5
displayMessage in SubClass called.
displayMessage in BaseClass called.
SubClass says: Hello!
```

Appendix 6. Code for ambiguous variable calls.

```
1  public class SuperClass {
2        private int X = 0;
3
4        public void setX() {
5              X = 1;
6        }
7  }
```

```
1  public class SubClass extends SuperClass {
2        private int X = 5;
3
4        public int getX() {
5              return X;
6        }
7  }
```

```
1  public class InheritanceTest {
2        public static void main(String[] args) {
3              SubClass C = new SubClass();
4              C.setX();
5              System.out.println(C.getX());
6        }
7  }
```