Week 4 Description

This week we'll work more with classes, discussing things such as static variables and methods, overloading, passing objects to methods, returning objects from methods, and *this* reference variable.

Static fields and methods

This week, we'll add more code to our BankAccount class.

The first thing we want to do is demonstrate static fields and methods. As we learned with the Math class back in week 1, static methods are called using the name of the class, and no object is needed to work with them.

static fields are a bit of a different concept. They are sometimes called class variables, and this name is a useful one for describing what they are. With a regular field, each object has its own copy of the field, and the field may have a different value in one object than it does in another. With a static field, there is only one copy of the field for the whole class, and all the objects share that same copy. If the field is public, it is referred to outside the class using the name of the class and the dot (.) operator, no object required. To summarize:

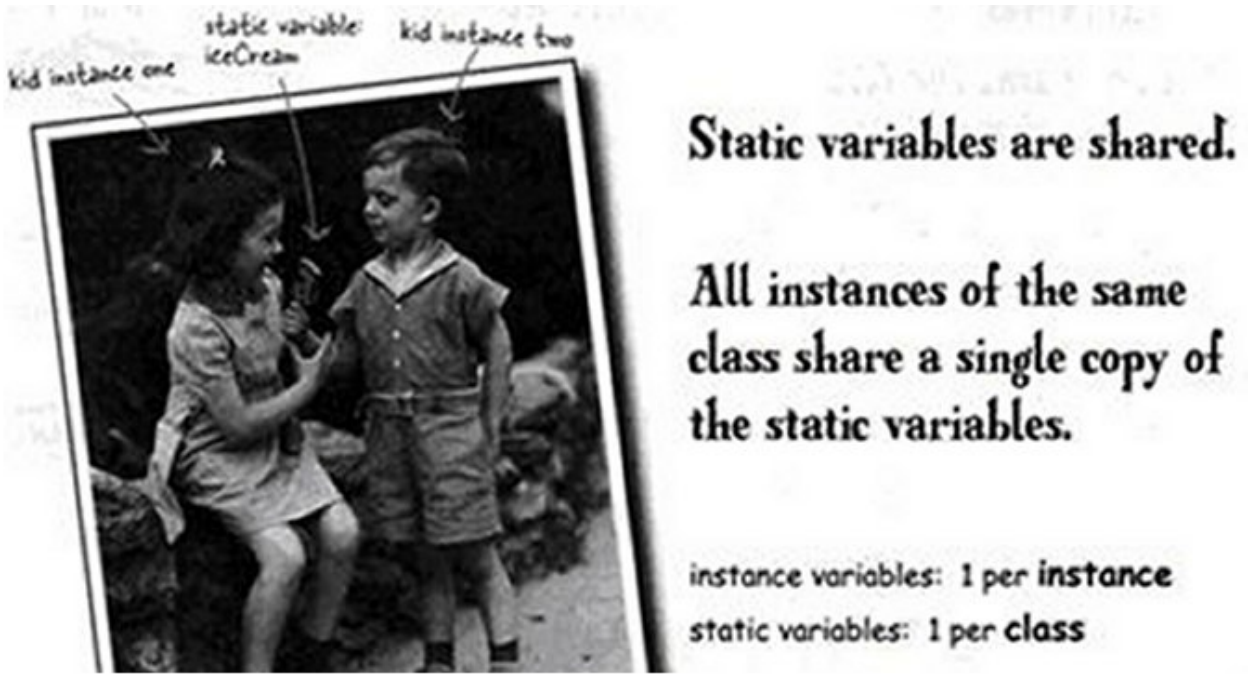
- Static fields are shared
- All instances of the same class share a single copy
- Static fields are used in the Tester class using the Class name rather than an object
- Static fields: 1 per class vs Instance variables (fields): 1 per instance

Our bank account class has a field for the balance and accountID. To automatically generate the accountID we need to make a static variable which will increment accountID everytime we make an object. Here is the BankAccount class after a new static variable. Note that NetBeans shows static fields in italics.

```
public class BankAccount {

    private int accountId;

    private double balance;

    private static int previousAccountID;    //static field

    public BankAccount(double bal){

        accountId = ++previousAccountID;

        balance = bal;    }
```

previousAccountID is a *static field* (sometimes called a *class field).* That means there is only one variable called previousAccountID for the entire BankAccount class. If we create 100 different BankAccount objects, they all share the same previousAccountID field.

balance is a *field* (because it does not have the word static in its declaration). Every BankAccount object that we create has its own balance field. The balance in myAccount is completely separate from the balance in yourAccount, because they are instance fields. In contrast, both accounts have the same previousAccountID, because this is a static field.



In the picture above, in case you can't tell, they are sharing an ice cream cone!

Counting number of instances

Let's create another static field in BankAccount class to count the number of instances created. This field is called howManyAccounts. Here is the code:

```
public class BankAccount {

    private static int howManyAccounts;

    public BankAccount(double bal){

                howManyAccounts++;

    }
```

```
    public static int getHowManyAccounts(){

        return howManyAccounts;

    }
```

In the code above, you can also see a static method. We put the word static before the return type of the method. We've seen these already with the Math class, remember?? However, this is the first time we've created one of our own. With a static method, you don't need an object of the class, you call it using the class name instead(Math.*round*()or BankAccount.*getHowManyAccounts*()). When you are coding them, there is a restriction: static methods can't use non-static variables (fields). This is because a static method isn't associated with an object, but an instance field is -- a static method has no object, so it can't use an object's instance fields.

Also, suppose our bank account pays interest on the balance. I am assuming that all bank accounts have the same interest rate. If so, it would be wasteful to have separate interest rate fields in each object. It would be better to have just one interest rate field for the entire class. We can do this by making a static constant. However, it is also possible that different accounts would have different interest rates – in this case, the interest rate would have to be a parameter to addInterest(), so that each bank account object would have its own interest rate.

Here is the complete class definition:

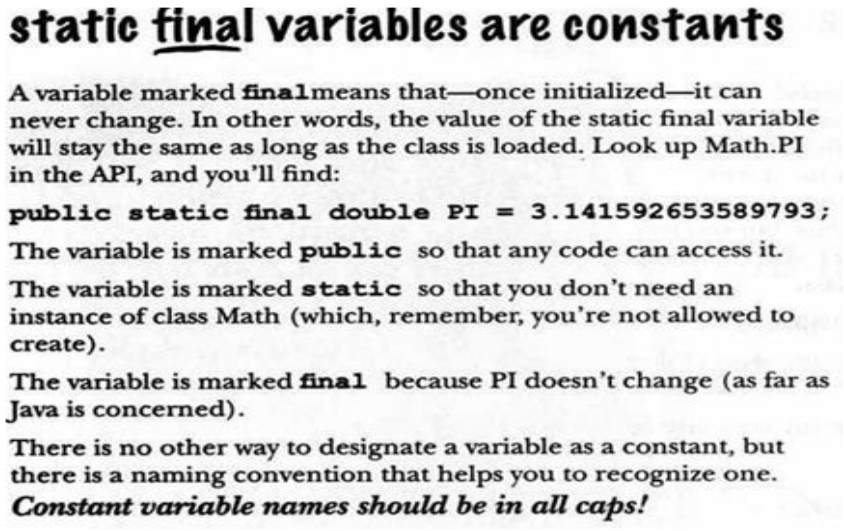
```
public class BankAccount{
private static int previousAccountID;
private static int howManyAccounts;
private double balance;
private int accountID;
private static final double INTEREST_RATE = 0.02;// stored as a fraction

public BankAccount(){
balance=0.0;
howManyAccount++;
}

public BankAccount(double initialBalance){
balance=initialBalance;
accountId = ++previousAccountID;
howManyAccount++;
}

public double getBalance(){
return balance;
}
```

```
public void deposit(double amount){
balance+=amount;
}

public void withdraw(double amount){
balance-=amount;
}

public void addInterest(){
balance+=balance * INTEREST_RATE;
}

//Method overloading
public void addInterest(double newRate){
balance+=balance * newRate;
}
}
```

final Fields

We could declare constants, by including the word final in the variable declaration. Often several methods of a class will make use of the same constants, so we can declare them as fields. Here too we use the keyword "final" to make them constant.

It would be a waste of memory to repeat the same constant in every instance of the class, so we usually make constant fields static. Remember, however, that not all static fields are constant. It is the keyword "final" that makes a field constant, not the keyword "static". Remember that, by convention, the names of final fields are ALL_UPPER_CASE, with underscores if the name is more than one word long.



static final variables are constants

A variable marked **final** means that—once initialized—it can never change. In other words, the value of the static final variable will stay the same as long as the class is loaded. Look up Math.PI in the API, and you'll find:

```
public static final double PI = 3.141592653589793;
```

The variable is marked **public** so that any code can access it.

The variable is marked **static** so that you don't need an instance of class Math (which, remember, you're not allowed to create).

The variable is marked **final** because PI doesn't change (as far as Java is concerned).

There is no other way to designate a variable as a constant, but there is a naming convention that helps you to recognize one.

Constant variable names should be in all caps!

Creating an object of main

The main is a static method. This means that all the rules of static methods apply: you can only use static variables inside them, etc. This can be pretty inconvenient at times.

So, instead, let's use an object to solve this problem. We'll create and initialize an object of the tester class in the main method, and move the code currently in main into the constructor for that object. By putting our code in the object's constructor rather than in the main method itself, we are using a non-static method to hold our code, which relieves the stress of the static method. The constructor will automatically be executed, as you know, when the new keyword is executed during object initialization. **Please use this technique from now on in your programs!!**

Method overloading

Overloaded methods are methods with the same name but different parameters. We have seen System.out.println, which is overloaded many times, once for each of the primitive data types. Here are the overloads of it from the Java API:

void	println() Terminates the current line by writing the line separator string.
void	println(boolean x) Prints a boolean and then terminate the line.
void	println(char x) Prints a character and then terminate the line.
void	println(char[] x) Prints an array of characters and then terminate the line.
void	println(double x) Prints a double and then terminate the line.
void	println(float x) Prints a float and then terminate the line.
void	println(int x) Prints an integer and then terminate the line.
void	println(long x) Prints a long and then terminate the line.
void	println(Object x) Prints an Object and then terminate the line.
void	println(String x) Prints a String and then terminate the line.

In our classes so far, we've only overloaded the constructor, but you can overload any method in a class. For example, we could overload the addInterest method to add

another version that has one parameter. Here is the original method followed by the overloaded method:

```
        public void addInterest() {
                balance *= INTEREST_RATE;
        }
        public void addInterest(double newRate) {
                balance *= newRate;
        }
```

You can see that the second version has a different parameter list, whereas the original version has no parameter. To call the first method, we could code:

account1.addInterest();

To call the overloaded version, we could code: account1.addInterest(.01);

Since the parameters are different, the JVM can tell which version of the method to execute.

Returning objects from methods

Just as we can return primitive types from a method, we can return an object from a method. To demonstrate, we have changed deposit method to return a BankAccount object. After changing balance, it creates a BankAccount object and then returns this new BankAccount object.

```
public BankAccount deposit(double money) {
        balance += money;
        return new BankAccount(balance);
    }
```

Objects as method arguments

Suppose we want BankAccount class to have a method that will transfer money from one account to another. If we make the method non-static, then it will have to operate on a BankAccount object, and we will specify that object when we call the method, by writing: objectName.methodName(…)

In this case, we want our method to act on *two* BankAccount objects. We can tell the method about the second object through the parameter list. We also need to tell it how much money to transfer, and we do this through the parameter list too. Here's one way to implement the method:

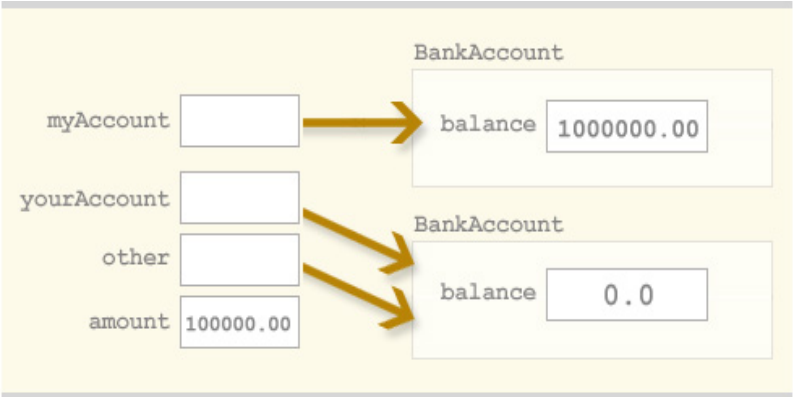
```
public void transfer(BankAccount other, double amount) {
  balance -= amount;
  other.balance += amount;
```

```
  }
```

The code below highlights the transfer method in action:

```
public class BankAccountTest {
  public static void main(String[] args) {
    // declare variables
    BankAccount myAccount;
    BankAccount yourAccount;

    // create objects
    myAccount = new BankAccount(1000000.0);
    yourAccount = new BankAccount(0.0);

    // print balances
    System.out.println("Before transfer:");
    System.out.printf("My balance is $%.2f\n",   myAccount.getBalance());
    System.out.printf("Your balance is $%.2f\n", yourAccount.getBalance());

    // transfer money from myAccount to yourAccount
    myAccount.transfer(yourAccount, 1000000.0);

    // print balances again
    System.out.println("\nAfter transfer:");
    System.out.printf("My balance is $%.2f\n", myAccount.getBalance());
    System.out.printf("Your balance is $%.2f\n",yourAccount.getBalance());
  }
}
```

When we call the transfer method, we prefix the method name with the BankAccount object myAccount, so when the method executes, it knows it is acting on the myAccount object. We pass yourAccount in the method call. The object yourAccount is assigned to the parameter "other" in the method definition, so "other" now refers to the same object as "yourAccount".



The transfer method is acting on two BankAccount objects. Each of these has a field called balance. We need to distinguish between the two balances. By default, when we say "balance" without any prefix in the method body, we mean the balance in the object on which the method was called. In this case, that is myAccount. To refer to the balance field in a **different** object, we need to explicitly say which object. In the example above, we do this by writing other.balance, since "other" is the variable referring to the second BankAccount in this method.

Here's another way to implement the transfer method:

```
public void transfer(BankAccount other, double amount) {
    withdraw(amount);
    other.deposit(amount);
}
```

When we call deposit, we prefix the method name with the object name "other", so the computer knows what object to deposit into. We have not prefixed the call to withdraw with any object name at all. Withdraw is a non-static method, so it needs to act on one object of the class where it is defined (BankAccount). How does it know which object to act on?

By default, if a method is called with no prefix, it acts on the same object as the method in which it is called. Here withdraw is called from inside transfer, so if transfer has been called to act on myAccount, then withdraw also acts on myAccount.

Here's another way to implement the transfer method:

```
public BankAccount transfer(BankAccount other, double amount) {
    withdraw(amount);
    other.deposit(amount);
    return other;
}
```

In this case, transfer method returns the balance of other object after transfer.

Keyword "this"

The BankAccount object that is called yourAccount in main is called "other" in the transfer method, because the value in yourAccount was assigned to the "other" variable through the method call. There is another BankAccount object, called myAccount in main. What is the name of this object inside the transfer method? It doesn't seem to have a name at all in that method.

In fact, it does – it is called "this". "this" is a Java keyword. Used inside a method, it refers to the object on which the method has been called.

Here are two more implementations of the transfer method:

```
public void transfer(BankAccount other, double amount) {
```

```
    this.balance -= amount;
    other.balance += amount;
}
public void transfer(BankAccount other, double amount) {
    this.withdraw(amount);
    other.deposit(amount);
}
```

In the methods above, you do not need to use “this”, but it's ok to use it if you like.

“this” cannot be used inside a static method, because a static method is not acting on any one object of a class.

“this” in constructors

Another use for “this” keyword is to call an overloaded constructor from another constructor of the same class. For example, our BankAccount class has two constructors:

```
public BankAccount(double myBalance) {
    balance = myBalance;
}
public BankAccount() {
    balance = 0.0;
}
```

we could rewrite these as

```
public BankAccount(double myBalance) {
   balance = myBalance;
}
public BankAccount() {
   this(0.0);
}
```

When the keyword “this” is used inside a constructor, followed by parentheses and an argument list, it means to call another overloaded version of the constructor of the same class, with a parameter list that matches the argument list. In the above example, we say “this” with one double argument, so the BankAccount constructor with one double parameter is called.

In the example above, there is not much advantage to using “this”, but if you have two constructors that do the same complex computation, the use of “this” can avoid having

to repeat large portions of source code. If you call an overloaded constructor using "this", the call **must be the *first* statement** in the body of your constructor.

"this" with variables

Local variables have scope from the place where they are defined to the end of the block in which they are defined. A local variable must be declared before it can be used.

In contrast, a field has scope throughout the class where it is defined.

We have seen that you cannot declare a local variable with the same name twice in the same block (this applies to method parameters, too). It is a syntax error to write

```
public class BankAccount {
    String foo = "Hello";
    for (int foo = O; …)
```

The compiler will complain that foo was already defined, and you are illegally trying to redefine it.

Similarly, you cannot declare two fields with the same name in the same class. However, you **can** declare a field and local variable (or a parameter) with the same name. When you use this name, by default the compiler will assume you mean the local variable or parameter, not the field. To indicate that you mean the field, you must prefix its name with "this".

For example, we would implement the BankAccount class and its constructor this way:

```
public class BankAccount {
    private double balance;
    public BankAccount(double balance) {
      this.balance = balance;
    }
}
```

In the constructor, "balance" has been used as the name of a parameter (which behaves like a local variable). It is also the name of a field of the class. When you use the name "balance" in the constructor, this means the local variable by default. To indicate that you do **not** mean the local variable, but the field, you must prefix the field name by "this".