Back to BankAccount:

Another important method: toString().

```
// returns String of info about this object
public String toString()
{
    return accNum + " " + name + " " + balance;
}
```

Back to Driver:

```
System.out.println(acct1);
```

(instead of: `System.out.println(acct1.toString());`)

Output: "$345 + Drumpt + 260"

---

Sometimes may want different versions of the same method:
ex: withdraw() that always withdraws $20

Method overloading - using the same name for multiple methods: (How will the compiler know which method to call?) => parameter lists must be different.
BankAccount:

```java
public class BankAccount {
    final double AMOUNT = 20;
    // value can't be changed

    public void withdraw() {
        this.balance -= AMOUNT;
    }

    AMOUNT = 30; // won't compile!
}
```

This was simplest approach, now more sophisticated.

Objects provide abstraction - we know they do something but we don't know how (implementation details).

ex: Scanner.nextInt() String: charAt()...

Only the object's class deals with implementation details.

Encapsulation - an object should be modifiable only by the object's methods.
Driver should not have direct access to object's data.

To enforce visibility modifiers:

- visibility modifiers - enforce a variable/method's accessibility by other classes:
  - public - accessible by any class within same project (by default).
  - private - only within object's class (another one we won't use): protected

To encapsulate a class:

- make all data variables private
- create public "setter" + "getter" methods for each variable
BankAccount now:

private String acctNum;

/** setter for acctNum */
public void setNum(String num) {
    this.acctNum = num;
}

/**
***/

/** getter for acctNum */
public String getNum() {
    return this.acctNum;
}

Driver now:

acct1.acctNum = "12345" \Rightarrow acct1.setNum("12345");
	println(acct1.getNum());

Why encapsulate?

1. Hide implementation:
   Suppose later decide acctNum should be int.
   With encapsulation, just need to change class, not driver.

2. Enforce validity:
   Suppose acctNum should be 9-digit.
   Can't enforce with acct1.acctNum = ...
   Can "  " acct1.setAccNum(...)
Discussed private data variables; methods may be private too.

Private methods accessible only by containing class (no outside class or Driver).

When useful? When method needs to be used only by the class.

Ex. BankAccount:

```java
// Checks that acctNum has length 9.
private boolean checkValid()
{
    if (acctNum.length == 9)
        return true;
    else
        return false;
}
```
Object Equality (not really in Book)

Parameter Passing ( )

**Primitive Data:**

```java
int num1; 1 - num1
int num2; 1 - num2
```

drawing:

```
num1 = 5; 5 num1
num2 = 10; 10 num2
```

primitive data: variable holds value.

objects: variable holds address of memory location

where object is stored

variable "points to" memory location of object

Suppose we had another constructor in BankAccount class

that takes all variables as a parameter

```java
public BankAccount (String num, String n, double bal) {
    this.acctNum = num;
    this.name = n;
    this.balance = bal;
}
```

```java
 Española <Sample Code>
```
In BankAccountDriver:

BankAccount acc1

acct1 = new BankAccount(“1234”, “Alice”, 100);
acct2 = new BankAccount(“5678”, “Bob”, 200);

(Want acc1 to have same info as acc2, try:

acct1 = acct2;

acct2.deposit(100);
print(acct2.getBalance()); => 300
print(acct1.getBalance()); => 300 But never deposited to acc1!!

How would we set the two accounts to be “equal”?

acct1.setName(acct2.getName());
acct1.setAcctNum(acct2.getAcctNum());
acct1.setBalance(acct2.getBalance());
Now, changes to acct2 won't affect acct1.

```plaintext
acct1
acct2
```
Parameter Passing

NumberSwap.java:

```java
    public class NumberSwap {
        public static void main(String[] args) {
            int num1, num2;
            num1 = 100;
            num2 = 200;
            swap(num1, num2);
            print(num1);
            print(num2);
        }

        public static void swap(int n1, int n2) {
            int temp = n1;
            n1 = n2;
            n2 = temp;
        }

        public static void print(int n) {
            System.out.println(n);
        }
    }
```

Output: num1: 100  num2: 200
Did not swap!

What happened?
Parameter passing - pass-by-value: parameter value is copied.

Primitive data - value of data copied.

Objects - "address"

Why? Since not just a single data value.

Now consider an object version of a number.

```java
Sample code: Number class

public class Number {
    int value;

    public Number(int v) {
        value = v;
    }
}
```

No encapsulation to keep example simple.
Add to Driver:

```java
public static void swap (Number n1, Number n2)
```

```java
int temp = n1.value;

n1.value = n2.value;

n2.value = temp;
```

Add to main():

```java
Number numb1 = new Number (100);
Number numb2 = new Number (200);

swap (numb1, numb2);

print (numb1, value);

print (numb2, value);
```

Output: 200
100

swaps!