Objects and Classes

The objectives of this chapter are:

- To discuss important aspects of the software development process
- To define objects and classes
  - To understand object models and class models
- To explain instance variables and instance methods
- To explain constructors

The Software Crisis

- We've all seen the statistics on software projects:
  - For every 6 new large scale systems put into production, 2 are cancelled.
  - The average software project overshoots its schedule by 50%
    - Large projects generally do worse
  - 75% of all systems are "operating failures" which either do not function as intended or are not used at all.

- One of the primary reasons for the software crisis is that the complexity of the required systems has outstripped the complexity management capabilities of structured (procedurally based) software paradigms.

The Object Oriented Paradigm

- One of the primary features of the O-O paradigm is its ability to manage complexity.
  - Clear identification of system entities (objects)
  - Abstract classification of system entities (classes)
  - Clear delineation of entity boundaries (roles, responsibilities, encapsulation, cohesion, coupling)
  - Clear separation of Abstraction boundaries (realistic class hierarchies)
  - Less ambiguous mapping between the real world and the software components modelling the real world. (objects)

- Unfortunately, the O-O paradigm is not clearly understood
  - Many programmers simply think that creating classes is OO programming
  - Many programmers erroneously think that because they use a language which supports OO concepts they are implementing OO designs
  - Many programmers, when confronted with code, cannot identify whether it is object-oriented and, if so, what elements make it OO.

The Object Oriented Paradigm

- Because the O-O paradigm is very much about complexity, it has shown to be difficult to teach and learn
  - Almost all examples are too simple or contrived
    - They illustrate a point, but have no real meaning
    - Real examples are expensive to create and require a large time commitment to learn (due to complexity)
  - Students are often left with the feeling, "why would I do this?"
  - When confronted with time pressures, programmers will often fall back to what they know best (usually, procedural programming techniques)
- The average programmer takes 6 months to 2 years to learn the O-O paradigm.
  - There are, generally speaking, no short-cuts
    - Review your progress! Keep all the code you've written and review
  - Requires abstract thinking
  - Most students report a time when, "the light goes on"
    - The "a-ha" experience

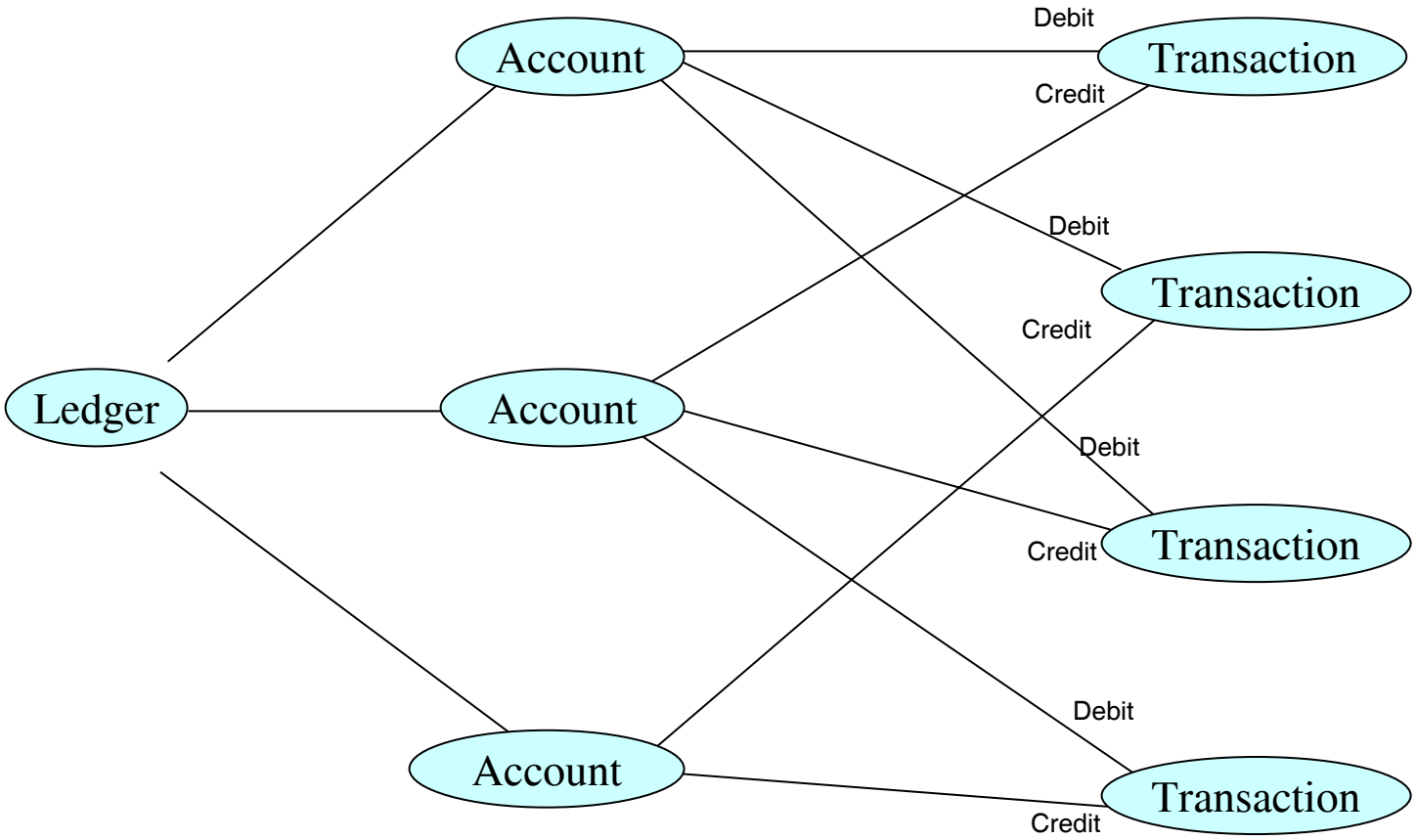
Where it all starts...

- The focus of the Object-Oriented paradigm is the Object
  - NOT classes. Classes are an abstraction of objects.
- Objects are identified from the real world
- An object model is created to represent the real world
  - The data associated with each object are identified and modelled.
  - The various behaviours of each object are identified and modelled.
  - Interactions between objects are modelled.
- The Objects in the object model are classified
  - Designers create classes
  - Programmers implement classes
  - Classes are logically grouped into a hierarchical classification structure
    - Because Java only allows single inheritance, that structure is a tree.
    - C++ allows multiple inheritance; the classification structure is a lattice.

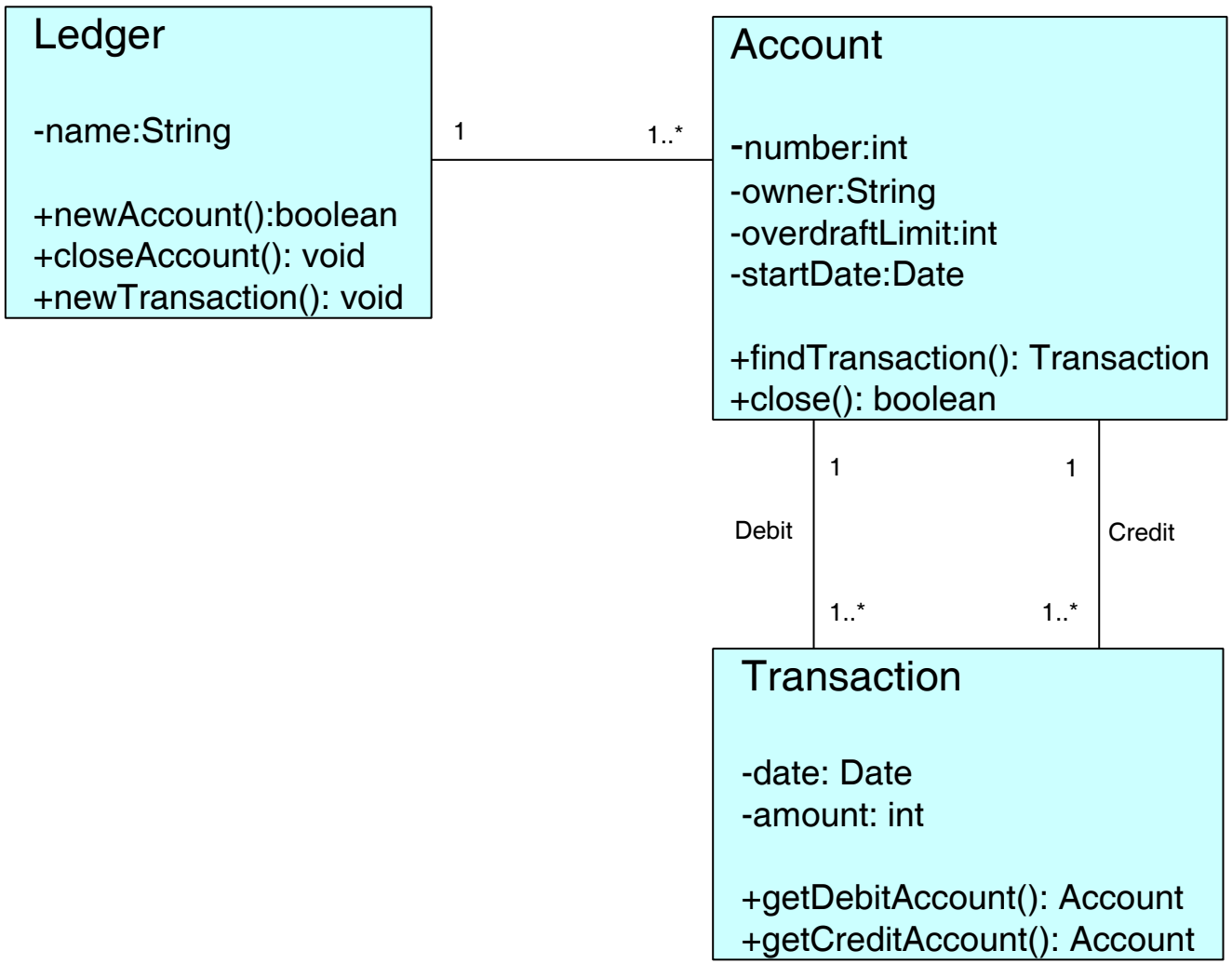
Creating Classes

- Objects in the object model are formalized
  - Objects are abstracted into classes
    - Only attributes and methods relevant to our domain are classified.
  - Attributes are formalized into instance variables
  - Behaviour is formalized into methods
  - Classes are represented on a class diagram
- Object interaction is also abstracted
  - Associations are identified
    - Added to class diagram
- Classes and the class diagram represent the static structure of the system
  - How the system behaves is not represented by this model.

Example Object Model



Example Class Model



Objects and Classes

- Programmers implement classes
  - Classes are templates or blueprints for Objects
  - Data and methods are defined within Classes
  - Classes must provide an implementation such that objects created from those classes behave as those defined in the Object model.
- An Object is the manifestation of a class
  - An object is an *Instance* of a class
  - The process of creating an object is called instantiation
  - The attributes of an object are called instance variables
  - The methods of an object are called instance methods
- In Java, Objects are created using the new keyword:

```
Employee anEmployee = new Employee();
```

Defining Classes

- A class definition must have the following:
  - The keyword "class" followed by the name of the class
  - The class body
- Before the keyword "class" is the optional modifier "public"
  - If a class is public, it must be defined within a file which is the same name as the class with a ".java" extension.
    - i.e. *Classname*.java
    - eg. HelloWorld.java, Account.java, Ledger.java, Transaction.java
    - most classes are declared *public*
- The class body contains:
  - Zero or more instance variables
  - Zero or more methods

Example Class Definition

in Employee.java:

Instance Variables:

Methods:

```
public class Employee
{
   String name;
   int salary;
   Date startingDate;
   [... more variable definitions ...]

   public int getSalary()
   {
         return salary;
   }
   public int computeHourlyRate()
   {
         // calculate hourly rate from salary
   }
   [... more method definitions ...]
}
```

Defining Instance Variables

- Instance variables are declared using the same syntax as ordinary variables.
  - Variables can be prefixed with a visibility modifier

```
modifier type variable_name;
```

- Variables can have one of 4 different visibilities:
  - public - the variable can be directly accessed from anywhere
  - private - the variable can only be directly accessed from within the class
  - protected - the variable can be access directly from within the class, within the package, or from within any subclass.
  - default (no modifier specified) - the variable can be accessed directly from within the package
- To preserve encapsulation, instance variables should be declared *private*.

Encapsulation

- Encapsulation is a very important O-O concept
  - Each object has 2 views. An internal view and an external view
- Encapsulation is a form of protection
  - Also called *Information Hiding*
  - The outside world does not have direct access to the internal implementation or representation of an object
  - As long as the external view does not change, the internal view can take on any form without affecting the outside world
  - By hiding data and providing methods to gain access to it, an object can maintain high data integrity
  - Methods have the responsibility of maintaining data integrity
- private visibility offers full encapsulation
  - protected and default offer limited encapsulation
  - public offers no encapsulation

Encapsulation Example

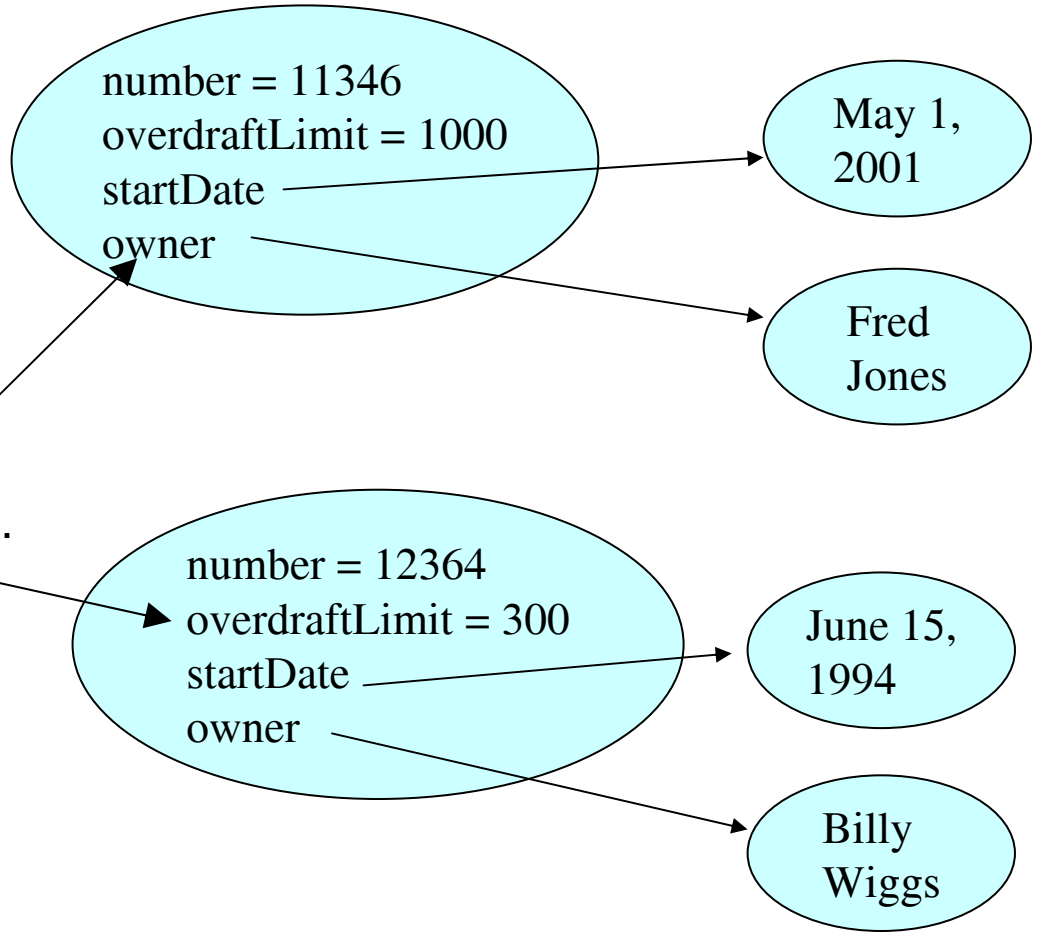
Class Definition:

```
public class Account
{
        private int number;
        private int
overdraftLimit;
        private Date startDate;
        private String owner;

[... methods ...]

}
```

Instances:



Instance variables are encapsulated.
- no direct access from outside the object

Each object has its own variables.
These variables are declared within the class.

Defining Instance Methods

- Method definitions include a method signature and a method body.
- Methods signatures are defined with the following syntax:

```
modifier return_type method_name(type name, ...)
```

- The return type can be:
  - a fundamental data type
  - an object reference
  - void (no return)
- Parameters are optional
  - If the method takes no parameters, empty brackets are required ()
  - Multiple parameters are separated by commas
  - Parameters are defined by type and name
    - A parameter is a local variable whose scope is the method.

Defining Instance Methods - Visibility

- Methods have the same visibility modifiers as variables
  - public - the method can be invoked from anywhere
  - private - the method can only be invoked from within the class
  - protected - the method can be invoked directly from within the class, within the package, or from within any subclass.
  - default (no modifier specified) - the method can be invoked directly from within the package
- If a method is part of the class's public interface (external view), the method should be public
- If a method is part of the class's internal implementation (ie, support method, etc), it should be private.
- Be careful using default or protected. Use only when justified.

Defining Instance Methods - Body

- A method's body contains all the statements to be executed as part of the method
- The method body is contained within curly braces after the method definition:
  - Use {} placement and indentation to clearly show code structure

```
public class CalculationSheet
{
    public void performCalculations()
    {
            [... method body ...]
    }

    public void clearSheet()
    {
    }
[...]
}
```

Returning values from methods

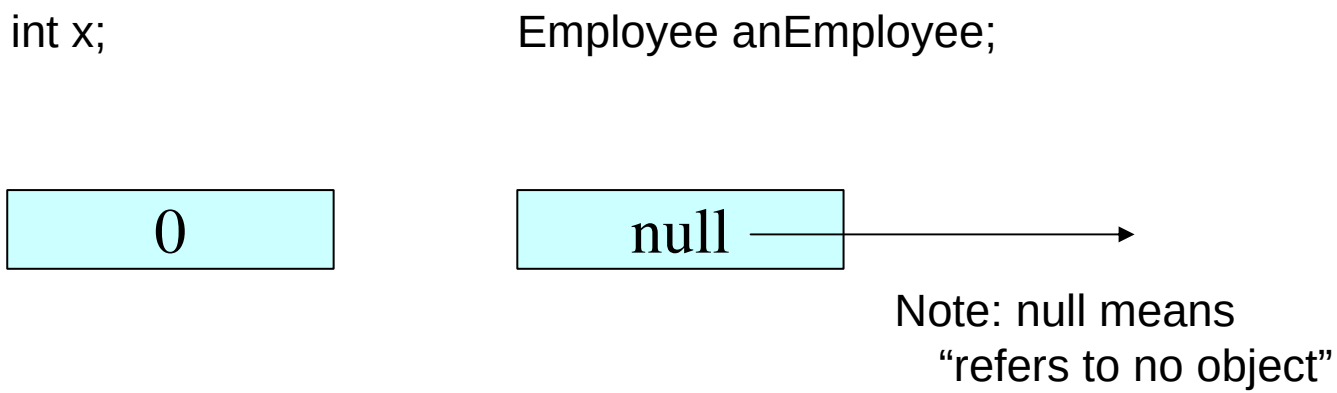
- A method which has a non-void return type MUST return a value
  - The return value's type must match the type defined in the method's signature.
  - A void method can use a return statement (with no return value) to exit the method.
  - The return value can be used the same as any other expression.

```
public class Car
{
    private int currentGear;
    private int currentRpms;

    public int calculateSpeed()
    {
            return currentRpms * currentGear;
    }
}
```

Classes as types

- When a class is defined, the compiler regards the class as a new type.
- When a variable is declared, its type can be a primitive type or "Class" type.
  - Any variable whose type is a class is an object reference.
    - The variable is a reference to an instance of the specified class.
    - The variables holds the address (in memory) of the object.



null References

- null means “refers to no object"
- Object references can be compared to null to see if an object is present or not.
- null is the default value of an object reference before it is initialized

```
Employee anEmployee;

[...]

   if (anEmployee == null)
   {
   }
```

Initializing Object References - new

- To initialize an object reference, you must assign it the address of an object

- The new operator creates a new instance and returns the address of the newly created object
  - new allocates memory for the object
  - new also invokes a method on the object called a constructor
  - new returns the address of the memory allocated for the object.

```
Employee anEmployee;

[...]

   anEmployee = new Employee();
```

Assigning Object References

- Assigning one reference to another results in two references to the same object
  - If two references contain the same memory address, they are referring to the same object.
  - Remember testing for equality of Strings using ==
- Each object has a reference count
  - When an object's reference count becomes zero, it will be collected by the garbage collector

```
Employee anEmployee = new Employee();
Employee anotherEmployee = anEmployee;
```

Invoking Instance Methods

- To invoke a method on an object, use the . (dot) operator

```
objectReference.methodName(parameters);
```

- If there is a return value, it can be used as an expression

```
Car aCar = new Car();

[...]

   if (aCar.calculateSpeed()>110)
   {
         System.out.println("You're Speeding!");
   }

[...]
```

Passing Parameters to Methods

- Method parameters are declared in the method's signature.
- When a method invocation is made, any parameters included in the invocation are passed to the method
  - All parameters are passed by value. Ie, a copy is made
    - The value of fundamental data types are copied
    - The value of object references (ie memory addresses) are copied
- Parameters become variables within the method. They are not known outside the method.

```
public float calculateInterestForMonth(float rate)
{
   return lowBalanceForMonth * (rate/12.0);
}
```

Overloading Methods

- Java allows for method overloading.
- A Method is overloaded when the class provides several implementations of the same method, but with different parameters
  - The methods have the same name
  - The methods have differing numbers of parameters or different types of parameters
  - The return type MUST be the same

```
public float calculateInterestForMonth()
{
    return lowBalanceForMonth * (defaultRate/12.0);
}

public float calculateInterestForMonth(float rate)
{
    return lowBalanceForMonth * (rate/12.0);
}
```

Accessor Methods - gets

- Objects have variables.
  - Because of encapsulation, those variables are generally private
  - However, the outside world may need to use those variables
  - The class implementor may choose to add a "get" method to return the value
- The usual name of the get method is the name of the variable prefixed with the word "get"
  - getName(), getAddress(), getPhone(), getBalance()

```
public class BankAccount
{
   private float balance;

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

Accessor Methods - sets

- Similarly, the outside world may need to set the value of an instance variable
  - The class implementor may choose to implement a set method.
  - The responsibility of the set method is to set the appropriate variable WHILST MAINTAINING data integrity of the object.
- The usual name of the set method is the name of the variable prefixed with the word "set"
  - setName(), setAddress(), setPhone(), setBalance()

```
public class BankAccount
{
   private String ownerName;

   public void setOwnerName(String aName)
   {
          ownerName = aName;
   }
```

Design Issues - When to provide gets and sets

- Get and set often provide confusion for novice programmers
  - Do all instance variables have them?
  - If so, why don't we make the instance variables public and access them directly?
  - Don't gets and sets violate encapsulation?
- Whether a variable has an associated get and set method is a design issue; it is not a coding issue.
- Imagine a BankAccount Class
  - All Bank Accounts have Account Numbers
  - Once an Account's Account Number has been set, should it be changeable?
- If we don't provide a set method, how do we initialize the variable in the first place?

Initializing Objects - Constructors

- When an object is created, all instance variables are initialized to the default value for their type
  - Fundamentals are 0, 0.0, '\000' or false
  - Object references are null
- In order to put the object into a usable state, its instance variables should be initialized to usable values
  - This could be accomplished by calling the various set methods
  - This is not always possible because it is not required that all instance variables have set methods.
- Java provides for another method of initializing objects
- When an object is created, a constructor is invoked. The responsibility of the constructor method is to initialize the object into a usable state.

Constructors

- Constructors have the following characteristics
  - There is NO return type. NOT even void
  - The method name is the same name as the class
  - Constructors can be overloaded
- In order to put the object into a usable state, its instance variables should be initialized to usable values
  - This could be accomplished by calling the various set methods
  - This is not always possible because it is not required that all instance variables have set methods.
- Java provides for another method of initializing objects
- When an object is created (using new), a constructor is invoked. The responsibility of the constructor method is to initialize the object into a usable state.

Constructors - Example

```
public class BankAccount
{
    String ownersName;
    int accountNumber;
    float balance;

    public BankAccount()
    {
    }

    public BankAccount(int anAccountNumber)
    {
            accountNumber = anAccountNumber;
    }

    public BankAccount(int anAccountNumber, String aName)
    {
            accountNumber = anAccountNumber;
            ownersName = aName;
    }

    [...]
}
```

Constructors - Example

- When an object is created (using new) the compiler determines which constructor is to be invoked by the parameters passed
  - Multiple constructors allows the class programmer to define many different ways of creating an object.

```
public static void main(String[] args)
{
    BankAccount anAccount = new BankAccount();
    BankAccount anotherAccount = new BankAccount(12345);
    BankAccount myAccount = new BankAccount(33423, "Craig");

}
```

Constructors

- If no constructors are defined for a class, the compiler automatically generates a default, no argument constructor
  - All instance variables are initialized to default values.
- However, if any constructor is defined which takes parameters, the compiler will NOT generate the default, no argument constructor
  - If you still need one, you have to explicitly define one.

Review

- What is the difference between classes and objects?
- What are the modifiers for classes, instance variables and methods? What do they mean?
- What is encapsulation? Why is it important?
- How are method parameters defined?
- How are method parameters passed?
- How do accessor methods support encapsulation?
- What are constructors?