

4B. Writing Classes II

- Objectives - when we have completed this chapter, you should be more familiar with:
 - Constant fields (public and private)
 - Invoking methods in the same class
 - Building a class incrementally
 - Testing a class
 - Writing a driver program

Constant Fields

- Recall that a class constant can be declared as follows:

```
private static final int MAX_AMOUNT = 5;
```

- Uses the static and final reserved words (static will be covered in detail later in the course)
- The name is in all caps with words separated by _
- In general, you do not want “magic numbers” (literal values) in your code.

Replace this code: `if (number > 5) . . .`

With this code:

```
if (number > MAX_AMOUNT)
```

Constant Fields

- Unlike variables, constants can be public without violating encapsulation.

- For example, the PI constant in the Math class.

```
double diameter = circumference * Math.PI;
```

- Referenced by using the name of the class, the dot operator, and the name of the constant.
 - When you define a constant inside your own class, you can reference directly without using the class name.

```
if (number > MAX_AMOUNT) . . .
```

Constant Fields

- Suppose the Student class had two types of students: undergrad and graduate.

```
public static final int GRADUATE = 0, UNDERGRAD = 1;
```

- Now client programs can set student type using constants (studentObj is an instance of Student):

Replace this code:

```
studentObj.setStudentType(0);
```

With this code:

```
studentObj.setStudentType(Student.GRADUATE);
```

Building a Class

- Suppose you wanted to create a class called Loan representing a loan with a balance, interest, and maximum loan amount
 - The balance starts at 0
 - The interest is 0.05 unless set otherwise
 - There are two loan types: employees and customers
 - Employees can borrow up to \$100,000
 - Customers can borrow up to \$10,000
- Create the empty class:

```
public class Loan {  
  
}
```

Building a Class

- The constructor for the bank class should take **an int parameter** representing the type of loan (employee or customer)
- Create an empty constructor; you will fill in the code later:

```
public class Loan {  
  
    public Loan(int accountType) {  
  
    }  
  
}
```

Building a Class

- Create an empty method for each of the following methods:

- A 'getter' method for the balance (a double):

```
public double getBalance() {  
  
}
```

- A 'setter' method for the interest rate (a double) that returns true and sets the interest only if the interest is from 0 to 1:

```
public boolean setInterest(double newInterest) {  
  
}
```

Building a Class

- Create an empty method for each of the following additional methods:
 - borrow: adds an additional amount to the loan and returns true only if the loan stays below the maximum amount.

```
public boolean borrow(double amount) {  
  
}
```

- calculateTotalBalance: returns a double representing the loan amount with the balance added.

```
public double calculateTotalBalance() {  
  
}
```


Building a Class

- Now you have method stubs for each one of your methods, but your program will not compile.
 - If a method does not have a void return type, then you must return something or it will not compile.
 - For now, just include a placeholder return – any value of that type. Examples:

```
public double getBalance() {  
    return 0;  
}
```

```
public boolean setInterest(double newInterest) {  
    return false;  
}
```

Building a Class

- You can now compile your program with empty methods. See the [Loan class in examples/method_stubs](#) for an example.
- This is the point where you would submit your project to the ungraded Web-CAT submission.

Building a Class: Variables

- Add instance variables to the class. Take another look at the class description:
 - Suppose you wanted to create a class called Loan representing a loan with a **balance**, **interest**, and **maximum loan amount**
 - **The balance starts at 0**
 - The interest is 0.05 unless set otherwise
 - There are two loan types: employees and customers
 - Employees can borrow up to \$100,000
 - Customers can borrow up to \$10,000

```
private double balance = 0, interestRate,  
    maxLoanAmount;
```

Building a Class: Constants

- Look for values that could be represented as constants:
 - **The interest is 0.05 unless set otherwise**
 - There are two loan types: employees and customers
 - **Employees can borrow up to \$100,000**
 - **Customers can borrow up to \$10,000**

The interest rate and maximum loan amount will be important for the class, but won't be needed by client programs. Therefore, the constants are private.

```
private static final double DEFAULT_INTEREST = 0.05;  
private static final double CUSTOMER_MAX = 10000,  
    EMPLOYEE_MAX = 100000;
```

Building a Class: Constants

- Look for values that could be represented as constants:
 - The interest is 0.05 unless set otherwise
 - **There are two loan types: employees and customers**
 - Employees can borrow up to \$100,000
 - Customers can borrow up to \$10,000

The loan type (customer or employee) will be set in the constructor using an int value. It would therefore be useful to provide constants for the client program:

```
public static final int EMPLOYEE_ACCOUNT = 0,  
    CUSTOMER_ACCOUNT = 1;
```

Building a Class: Constructor

- The constructor should set the interest rate to the default interest and then set the maximum loan amount based on the parameter.
- To test your constructor, instantiate objects in the interactions pane and check the values of the instance variables in the workbench (next slide).

Building a Class: Constructor

- The constructor should set the interest rate to the default interest and then set the maximum loan amount based on the parameter.
 - To test your constructor, instantiate objects in the interactions pane and check the values of the instance variables in the workbench.

```
Loan customer = new Loan(Loan.CUSTOMER_ACCOUNT);
```

```
Loan emp1 = new Loan(Loan.EMPLOYEE_ACCOUNT);
```

Building a Class: Methods

- You can now complete your getBalance method and setInterest method.
 - In general, **get methods will not change the state of the object**, but will only return a value.
 - Set methods will often have a boolean return type to indicate if the inputs are valid and the method actually “set” the field.
 - Test each one of your methods as you create them. You can view each method’s effect on instance variables using interactions and the workbench.

Building a Class: Testing

- Example: Test both the function and the return of the setInterest method (workbench and interactions).

```
Loan loanObj = new Loan(Loan.CUSTOMER_ACCOUNT);  
loanObj.setInterest(-1);  
loanObj.setInterest(-1)  
false  
loanObj.setInterest(2)  
false  
loanObj.setInterest(0.5)  
true
```

See the completed Loan class in [Loan.java](#)

Repeated Code

- In general, you do not want to repeat calculations in your code.
 - The toString method should return a string that includes the total balance.
 - In this case, make a call to the method that performs the calculation:

```
public String toString() {  
    String output = "Loan amount: "  
        + balance + "\r\n"  
        + "Total balance with interest: $"  
        + calculateTotalBalance()  
        + "\r\n";  
    return output;  
}
```

Driver Program

- The Loan class cannot be run as an application, but can only be used by other programs.
- In this case, we might want to create a driver program that user's can execute to set up a loan, borrow an amount, and view their balance.
- See [LoanCalculator.java](#) for an example of a driver program for the Loan class.