MC322 - Object Oriented Programming
Lesson 4.1
Advanced Object-Oriented Concepts

Prof. Marcos M. Raimundo
Instituto de Computação - UNICAMP

Before we embark

- Advanced Object-Oriented (OO) Concepts:
  - Constructors
  - Operator overloading
  - Multiple inheritance
- Error-handling techniques
- Understanding how scope applies to object-oriented design
- Importance of grasping these concepts:
  - Not vital at a high-level understanding of OO design
  - Essential for anyone involved in design and implementation

Constructors

Constructors

- Constructors are a new concept for people doing structured programming.
- They do not exist in non-OO languages such as COBOL, C, and Basic.
- In object-oriented (OO) languages, constructors are methods that:
  - Share the same name as the class.
  - Have no return type.
- Example of a constructor in a Java-like syntax for a class named `Cabbie`:

```
public Cabbie() {
    /* code to construct the object */\\
}
```

- The compiler recognizes a method as a constructor if its name is identical to the class name.

Caution

Note again that a constructor does not have a return value. If you provide a return value, the compiler will not treat the method as a constructor.

Example

If you include the following code in the class, the compiler will not consider this a constructor because it has a return value—in this case an integer:

```
public int Cabbie(){
    /* code to construct the object */
}
```

This syntax requirement can cause problems because this code will compile but will not behave as expected.

When Is a Constructor Called?

When a new object is created, one of the first things that happens is that the constructor is called. Check out the following code:

```
Cabbie myCabbie = new Cabbie();
```

The `new` keyword creates a new instance of the `Cabbie` class, thus allocating the required memory. Then the constructor itself is called, passing the arguments in the parameter list. The constructor provides the developer the opportunity to attend to the appropriate initialization.

Thus, the code `new Cabbie()` will instantiate a `Cabbie` object and call the `Cabbie` method, which is the constructor.

What's Inside a Constructor?

Perhaps the most important function of a constructor is to initialize the memory allocated when the `new` keyword is encountered. In short, code included inside a constructor should set the newly created object to its initial, stable, safe state.

For example, if you have a counter object with an attribute called `count`, you need to set `count` to zero in the constructor:

```
count = 0;
```

The Default Constructor

If you write a class and do not include a constructor, the class will still compile, and you can still use it. If the class provides no explicit constructor, a default constructor will be provided. It is important to understand that at least one constructor always exists, regardless of whether you write a constructor yourself. If you do not provide a constructor, the system will provide a default constructor for you.

Besides the creation of the object itself, the only action that a default constructor takes is to call the constructor of its superclass. In many cases, the superclass will be part of the language framework, like the `Object` class in Java. For example, if a constructor is not provided for the `Cabbie` class, the following default constructor is inserted:

```
public Cabbie(){
    super();
}
```

If you were to de-compile the bytecode produced by the compiler, you would see this code.The compiler actually inserts it.

Default Constructor Considerations

If `Cabbie` does not explicitly inherit from another class, the `Object` class will be the parent class. Perhaps the default constructor might be sufficient in some cases; however, in most cases, some memory initialization should be performed. Regardless of the situation, it is good programming practice always to include at least one constructor in a class. If there are attributes in the class, initializing them is always good practice.

Providing a Constructor

The rule of thumb is always to provide a constructor, even if you do not plan on doing anything inside it. You can provide a constructor with nothing in it and add to it later. Although there is technically nothing wrong with using the default constructor provided by the compiler, it is always nice to know exactly what your code looks like.

It is not surprising that maintenance becomes an issue here. If you depend on the default constructor and maintenance is performed on the class that added another constructor, then the default constructor is not created. In short, the default constructor is only added if you don't include one. The default constructor is not included as soon as you include just one.

Using Multiple Constructors

In many cases, an object can be constructed in more than one way. To accommodate this situation, you need to provide more than one constructor.

```
public class Count {
    int count;
    public Count(){
        count = 0;
    }
}
```

On the one hand, we want to initialize the attribute `count` to zero. We can easily accomplish this by having a constructor initialize `count` to zero as follows:

```
public Count(){
    count = 0;
}
```

On the other hand, we might want to pass an initialization parameter that allows `count` to be set to various numbers:

```
    public Count (int number){
        count = number;
    }
```

This is called overloading a method (overloading pertains to all methods, not just constructors).

Overloading Methods

Overloading allows a programmer to use the same method name over and over, as long as the signature of the method is different each time. The signature consists of the method name and a parameter list.

Thus, the following methods all have different signatures:

```
public void getCab();
// different parameter list

public void getCab(String cabbieName);
// different parameter list

public void getCab(int
     numberOfPassengers);
```

Signatures

Depending on the language, the signature may or may not include the return type. In Java and C#, the return type is not part of the signature. For example, the following methods would conflict even though the return types are different:

```
public void getCab(String cabbieName);
public int getCab(String cabbieName);
```

The best way to understand signatures is to write some code and run it through the compiler.

DataBaseReader Class Constructors - Code

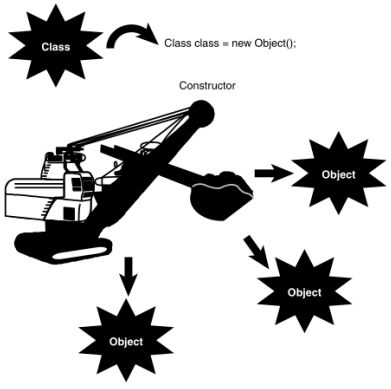
```
public class DataBaseReader {
    String dbName;
    int startPosition;

    // initialize just the name
    public DataBaseReader(String name){
        dbName = name;
        startPosition = 0;
    };

    // initialize the name and the position
    public DataBaseReader(String name, int pos){
        dbName = name;
        startPosition = pos;
    };

    .. // rest of class
}
```

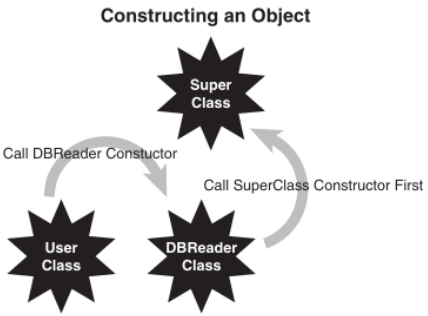
Note how `startPosition` is initialized in both cases. If the constructor does not pass the information via the parameter list, it is initialized to a default value, like 0.



Creating a new object.

Constructing the Superclass

When using inheritance, you must know how the parent class is constructed. How a constructor is inherited is not as obvious. After the `new` keyword is encountered and the object is allocated, the following steps occur:



1. The first thing that happens inside the constructor is that the constructor of the class's superclass is called. If there is no explicit call to the superclass constructor, the default is called automatically; however, you can see the code in the bytecodes.
2. Then, each class attribute of the object is initialized. These attributes are part of the class definition (instance variables), not the attributes inside the constructor or any other method (local variables).
3. Then the rest of the code in the constructor executes.

The Design of Constructors

As we have already seen, when designing a class, it is good practice to initialize all the attributes. In some languages, the compiler provides some sort of initialization. As always, don't count on the compiler to initialize attributes! In Java, you cannot use an attribute until it is initialized. If the attribute is first set in the code, make sure that you initialize the attribute to some valid condition—for example, set an integer to zero.

Constructors are used to ensure that the application is in a stable state (I like to call it a "safe" state). For example, initializing an attribute to zero, when it is intended for use as a denominator in a division operation, might lead to an unstable application. You must take into consideration the fact that a division by zero is an illegal operation. Initializing to zero is not always the best policy.

During the design, it is good practice to identify a stable state for all attributes and then initialize them to this stable state in the constructor.

Error Handling

Handling Errors

It is rare for a class to be written perfectly the first time. In most, if not all, situations, things will go wrong. Assuming that your code can detect and trap an error condition, you can handle the error in several different ways.

On page 223 of their book "Java Primer Plus," Tyma, Torok, and Downing state:

- Fix it
- Ignore the problem by squelching it
- Exit the runtime in some graceful manner

On page 139 of their book "Object-Oriented Design in Java," Gilbert and McCarty state:

- Ignore the problem—not a good idea!
- Check for potential problems and abort the program when you find a problem
- Check for potential problems, catch the mistake, and attempt to fix the problem
- Throw an exception (often this is the preferred way to handle the situation)

Ignoring the Problem

- Simply ignoring a potential problem is a recipe for disaster.
- And if you are going to ignore the problem, why bother detecting it in the first place?
- The bottom line is that you should not ignore the problem.
- The primary directive for all applications is that the application should never crash.
- If you do not handle your errors, the application will eventually terminate ungracefully or continue in a mode that can be considered an unstable state.
- In the latter case, you might not even know you are getting incorrect results for some period of time.

Checking for Problems and Aborting the Application

- If you choose to check for potential problems and abort the application when a problem is detected, the application can display a message indicating that there is a problem.
- In this case, the application gracefully exits, and the user is left staring at the computer screen, shaking her head and wondering what just happened.
- Although this is a far superior option to ignoring the problem, it is by no means optimal.
- However, this does allow the system to clean up things and put itself in a more stable state, such as closing files.

Checking for Problems and Attempting to Recover

- Checking for potential problems, catching the mistake, and attempting to recover is a far superior solution than simply checking for problems and aborting.
- The following code uses an if statement to avoid system exceptions because you cannot divide by zero. By catching the exception and setting the variable a to 1, at least the system will not crash. However, setting a to 1 might not be a proper solution.

```
if (a == 0)
    a = 1;
c = b/a;
```

- It is hard to determine where a problem first appears. And it might take a while for the problem to be detected. It is important to design error handling into the class right from the start.

A Mix of Error Handling Techniques

Even though this type of error handling is not necessarily object-oriented, it has a valid place in OO design. Throwing an exception (discussed in the next section) can be expensive in terms of overhead. Thus, although exceptions are a great design choice, you will still want to consider other error-handling techniques, depending on your design and performance needs.

Throwing an Exception

Most OO languages provide a feature called exceptions. In the most basic sense, exceptions are unexpected events within a system. Exceptions provide a way to detect problems and then handle them. In Java, C#, and C++, exceptions are handled by the keywords `catch` and `throw`. This might sound like a baseball game, but the key concept here is that a specific block of code is written to handle a specific exception. This solves the problem of trying to figure out where the problem started and unwinding the code to the proper point.

Here is the structure for a try/catch block:

```
try {
    // possible nasty code
} catch(Exception e) {
    // code to handle the exception
}
```

Exception Handling in Try/Catch Blocks

If an exception is thrown within the try block, the catch block will handle it. When an exception is thrown while the block is executing, the following occurs:

1. The execution of the try block is terminated.
2. Catch clauses are checked to determine whether an appropriate catch block for the offending exception was included: Might have more than one catch clause per try block.
3. If none of the catch clauses handle the offending exception, it is passed to the next higher-level try block. (If the exception is not caught in the code, the system ultimately catches it, and the results are unpredictable, i.e., an application crash.)
4. If a catch clause is matched (the first match encountered), the statements in the catch clause are executed.
5. Execution then resumes with the statement following the try block.

Exception Granularity

You can catch exceptions at various levels of granularity. You can catch all exceptions or just check for specific exceptions, such as arithmetic exceptions. If your code does not catch an exception, the Java runtime will—and it won't be happy about it!

Exception Handling Example in Java

```
try {
    // possible nasty code
    count = 0;
    count = 5 / count;
} catch(ArithmeticException e) {
    // code to handle the exception
    System.out.println(e.getMessage());
    count = 1;
}
System.out.println("The exception is handled.");
```

Constructing an object.

- Division by zero (because count is equal to 0) within the try block will cause an arithmetic exception.
- If the exception were generated (thrown) outside a try block, the program would most likely have been terminated.
- Being within a try block, the catch block is checked to see whether the specific

You can catch all exceptions by using the following code:

```
try {
    // possible nasty code
} catch(Exception e) {
    // code to handle the exception
}
```

The Concept of Scope

Objects and Attributes

- Multiple objects can be instantiated from a single class. Each of these objects has a unique identity and state. This is an important point. Each object is constructed separately and is allocated its separate memory. However, if properly declared, some attributes and methods may be shared by all the objects instantiated from the same class, thus sharing the memory allocated for these class attributes and methods.
- Methods represent the behaviors of an object; attributes represent the object's state. There are three types of attributes:
  - Local attributes
  - Object attributes
  - Class attributes

A Shared Method

A constructor is a good example of a method that is shared by all instances of a class.

Local Attributes

Local attributes are owned by a specific method. Consider the following code:

```
public class Number {
    public method1() {
        int count;
    }
    public method2() {
    }
}
```

The method `method1` contains a local variable called `count`. This integer is accessible only inside `method1`. The method `method2` has no idea that the integer `count` even exists. At this point, we introduce a very important concept: scope. Attributes (and methods) exist within a particular scope. In this case, the integer `count` exists within the scope of `method1`.

The class itself has its own scope. Each instance of the class (that is, each object) has its own scope. Both method1 and method2 have their own scopes as well. Because count lives within method1's curly braces, when method1 is invoked, a copy of count is created. When method1 terminates, the copy of count is removed.

Scope in Methods

For some more fun, look at this code:

```
public class Number {
    public method1() {
        int count;
    }
    public method2() {
        int count;
    }
}
```

In this example, there are two copies of an integer `count` in this class. Remember that `method1` and `method2` each has its own scope. Thus, the compiler can tell which copy of `count` to access simply by recognizing which method it is in. You can think of it in these terms:

```
method1.count;
method2.count;
```

As far as the compiler is concerned, the two attributes are easily differentiated, even though they have the same name. It is almost like two people having the same last name, but based on the context of their first names, you know that they are two separate individuals.

Class Attributes

Consider the following code:

```
public class Number {
    int count; // available to both method1 and method2

    public method1() {
        count = 1;
    }

    public method2() {
        count = 2;
    }
}
```

In this case, the class attribute `count` is declared outside the scope of both `method1` and `method2`. However, it is within the scope of the class. Thus, `count` is available to both `method1` and `method2`. Notice that the code for both methods is setting `count` to a specific value. There is only one copy of `count` for the entire object, so both assignments operate on the same copy in memory. However, this copy of `count` is not shared between different objects.

Class Attributes

To illustrate, let's create three copies of the `Number` class:

```
Number number1 = new Number();
Number number2 = new Number();
Number number3 = new Number();
```

Each of these objects—number1, number2, and number3—is constructed separately and is allocated its own resources.There are actually three separate instances of the integer count.When number1 changes its attribute count, this in no way affects the copy of count in object number2 or object number3. In this case, integer count is an object attribute.

Playing with Scope

You can play some interesting games with scope. Consider the following code:

```
public class Number {
    int count;
    public method1() {
        int count;
    }
    public method2() {
        int count;
    }
}
```

In this case, there are actually three totally separate memory locations with the name of `count` for each object. To access the object variable from within one of the methods, say `method1()`, you can use a pointer called `this` in the C-based languages:

```
public method1() {
    int count;
    this.count = 1;
}
```

Notice that there is some code that looks a bit curious:

```
this.count = 1;
```

The selection of the word `this` as a keyword is perhaps unfortunate. However, we must live with it. The use of the `this` keyword directs the compiler to access the object variable `count` and not the local variables within the method bodies.

Note

The keyword `this` is a reference to the current object.

Operator Overloading

Operator Overloading

Some OO languages allow you to overload an operator. C++ is an example of one such language. Operator overloading allows you to change the meaning of an operator. For example, when most people see a plus sign, they assume it represents addition. If you see the equation

$$X = 5 + 6;$$

you expect that X would contain the value 11. And in this case, you would be correct. However, there are times when a plus sign could represent something else. For example, in the following code:

```
String firstName = "Joe", lastName = "Smith";
String Name = firstName + " " + lastName;
```

You would expect that `Name` would contain "Joe Smith". The plus sign here has been overloaded to perform string concatenation.

String Concatenation

String concatenation is when two separate strings are combined to create a new, single string.

Operator Overloading

In the context of strings, the plus sign does not mean addition of integers or floats, but concatenation of strings. What about matrix addition? You could have code like this:

```
Matrix a, b, c;
c = a + b;
```

Thus, the plus sign now performs matrix addition, not addition of integers or floats. Overloading is a powerful mechanism. However, it can be downright confusing for people who read and maintain code. In fact, developers can confuse themselves. To take this to an extreme, it would be possible to change the operation of addition to perform subtraction. Why not?

More recent OO languages like Java and .NET do not allow operator overloading. While these languages do not allow the option of overloading operators, the languages themselves do overload the plus sign for string concatenation, but that's about it. The designers of Java must have decided that operator overloading was more of a problem than it was worth.

Multiple Inheritance

Multiple Inheritance

We cover inheritance in much more detail in Chapter 7, "Mastering Inheritance and Composition." However, this is a good place to begin discussing multiple inheritance, which is one of the more powerful and challenging aspects of class design.

As the name implies, multiple inheritance allows a class to inherit from more than one class. In practice, this seems like a great idea. Objects are supposed to model the real world, are they not? And there are many real-world examples of multiple inheritance. Parents are a good example of multiple inheritance. Each child has two parents—that's just the way it is. So it makes sense that you can design classes by using multiple inheritance. In some OO languages, such as C++, you can.

Multiple Inheritance Complexity

However, this situation falls into a category similar to operator overloading. Multiple inheritance is a very powerful technique, and in fact, some problems are quite difficult to solve without it. Multiple inheritance can even solve some problems quite elegantly. However, multiple inheritance can significantly increase the complexity of a system, both for the programmer and the compiler writers.

As with operator overloading, the designers of Java and .NET decided that the increased complexity of allowing multiple inheritance far outweighed its advantages, so they eliminated it from the language. In some ways, the Java and .NET language construct of interfaces compensates for this; however, the bottom line is that Java and .NET do not allow conventional multiple inheritance.

Behavioral and Implementation Inheritance

Java and .NET interfaces are a mechanism for behavioral inheritance, whereas abstract classes are used for implementation inheritance. The bottom line is that Java and .NET interfaces provide interfaces, but no implementation, whereas abstract classes may provide both interfaces and implementation. This topic is covered in great detail in Chapter 8, "Frameworks and Reuse: Designing with Interfaces and Abstract Classes."

Object Operations

Complexity of Copying and Comparing Objects

- Basic operations in programming become more complicated with complex data structures and objects.
- Copying or comparing primitive data types is straightforward.
- Copying and comparing objects is more complex.
- Scott Meyers devotes an entire section to copying and assigning objects in his book "Effective C++".

Classes and References

The problem with complex data structures and objects is that they might contain references. Simply making a copy of the reference does not copy the data structures or the object that it references. In the same vein, when comparing objects, simply comparing a pointer to another pointer only compares the references—not what they point to.

Complexity of Copying and Comparing Objects

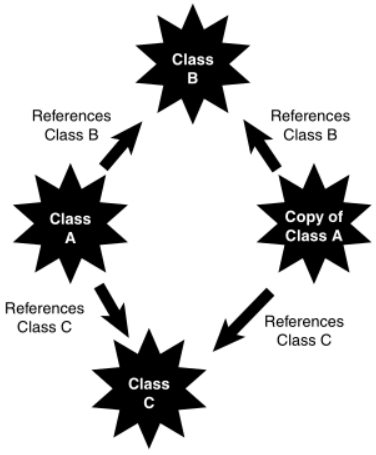
- The problems arise when comparisons and copies are performed on objects. Specifically, the question boils down to whether you follow the pointers or not.
- Regardless, there should be a way to copy an object. Again, this is not as simple as it might seem.
- Because objects can contain references, these reference trees must be followed to do a valid copy (if you truly want to do a deep copy).

Deep Versus Shallow Copies

A deep copy is when all the references are followed and new copies are created for all referenced objects. There might be many levels involved in a deep copy. For objects with references to many objects, which in turn might have references to even more objects, the copy itself can create significant overhead. A shallow copy would simply copy the reference and not follow the levels. Gilbert and McCarty have a good discussion about what shallow and deep hierarchies are on page 265 of Object-Oriented Design in Java in a section called "Prefer a Tree to a Forest."

Shallow and deep copy

- If you just do a simple copy of the object (called a bitwise copy), any object that the primary object references will not be copied: only the references will be copied. Example in Figure.
- To perform a complete copy, in which all reference objects are copied, you have to write the code to create all the sub-objects.
- This problem also manifests itself when comparing objects. Because objects contain references, these reference trees must be followed to do a valid comparison of objects.
- In most cases, languages provide a default mechanism to compare objects. As is usually the case, do not count on the default mechanism. When designing a class, you should consider providing a comparison function in your class that you know will behave as you want it to.



Following object references.

Conclusion

Conclusion This chapter covered a number of advanced OO concepts that, although perhaps not vital to a general understanding of OO concepts, are quite necessary in higher-level OO tasks, such as designing a class.

MC322 - Object Oriented Programming
Lesson 4.1
Advanced Object-Oriented Concepts

Prof. Marcos M. Raimundo
Instituto de Computação - UNICAMP