CSC 1051 Algorithms & Data Structures I

DESIGNING CLASSES

Class Anatomy

A class that contains a static main method represents the driver of a program

Let's look at a class that defines a classic object (with state and behavior)

The anatomy of a class includes

- instance data
- constructor(s)
- methods

Class Anatomy

A *tally counter* is a device that helps you count people (as they enter a room, for example)

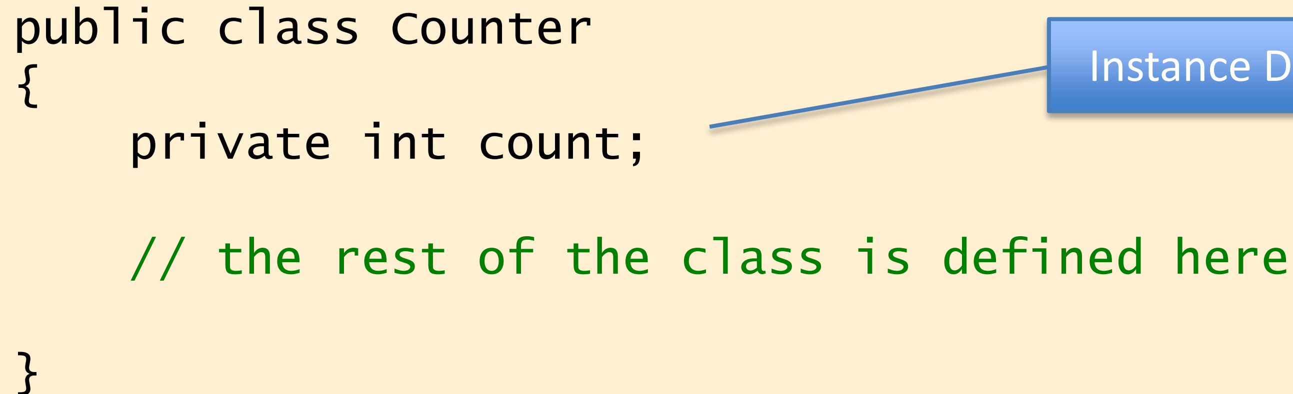
It's used to track the number of occupants at an event

It has a button that increments a counter every time it's clicked, and a way to reset that counter to 0

We will model the idea of a tally counter in software by defining a class called Counter

Class Anatomy

The only data defined in Counter is a single int variable



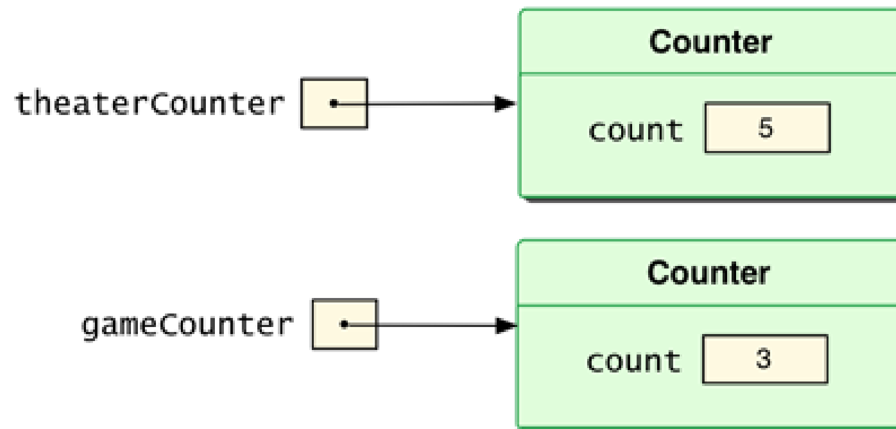

```java
public class Counter
{
    private int count;

    // the rest of the class is defined here

}
```

The variable count is called instance data, because each instance of the class (each object) has its own

Every time a Counter object is created, memory space is reserved to store the count for that object

Class Anatomy

Instance data allows each object to have its own state



Remember, a class is the pattern from which an object is made

The variable is declared once in the class, representing the integer in each Counter object

Class Anatomy

A variable declared at the class level can be accessed by any method in the class

Declaring the count variable as private keeps other objects from directly accessing it

That way, the value of count is only changed by methods of the Counter class

This is the basis of encapsulation

Class Anatomy

The public methods of the Counter class define its behavior

They are the services that a Counter object will perform when they are called by another object

```
// Increments the counter when "clicked"
public void click()
{
    count++;
}
```

The click method takes no parameters and returns no value

Class Anatomy

Two more Counter methods:

```
// Resets the count back to 0
public void reset()
{
    count = 0;
}

// Returns the current count of this Counter
public int getCount()
{
    return count;
}
```

Class Anatomy

```
// Returns the current count as a string
public String toString()
{
    return count + "";
}
```

It's often a good idea to define a toString method for a class

toString is automatically called when you print an object or concatenate an object to a string

Concatenating count to the the empty string is a quick way to convert the number to a string

Class Anatomy

A constructor is like a special method that is called when an object is created

```
// Initializes the count to 0
public Counter()
{
    count = 0;
}
```

A constructor sets up a newly created object

It has no return type (not even void) and has the same name as the class

Example: Dice

Let's examine another class that defines a classic object

A die (singular of dice) might have the traditional six sides, or it may be more specialized

So the instance data of a Die class would track how many sides a die has and what value is currently showing

Explore **Example: Dice** in Rephactor.

Unified Modeling Language (UML)

UML stands for the *Unified Modeling Language*

UML diagrams show relationships among classes and objects

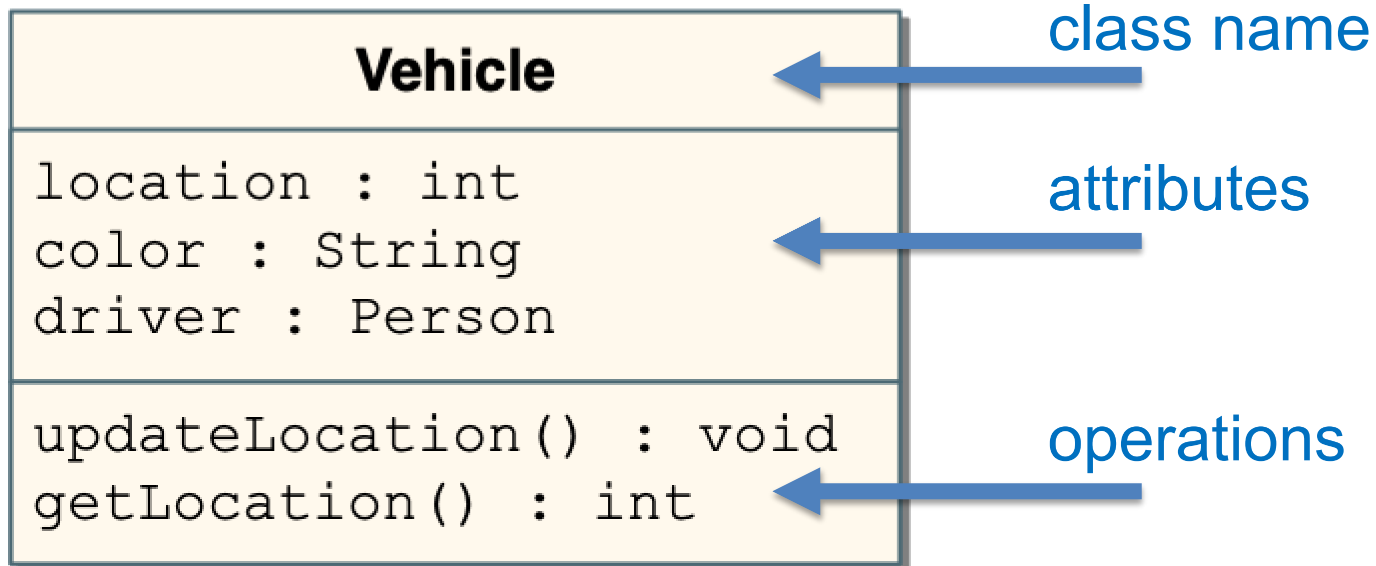
A UML *class diagram* consists of one or more classes, each with sections for the class name, attributes (data), and operations (methods)

Lines between classes represent *associations*

A dotted arrow shows that one class *uses* the other (calls its methods)

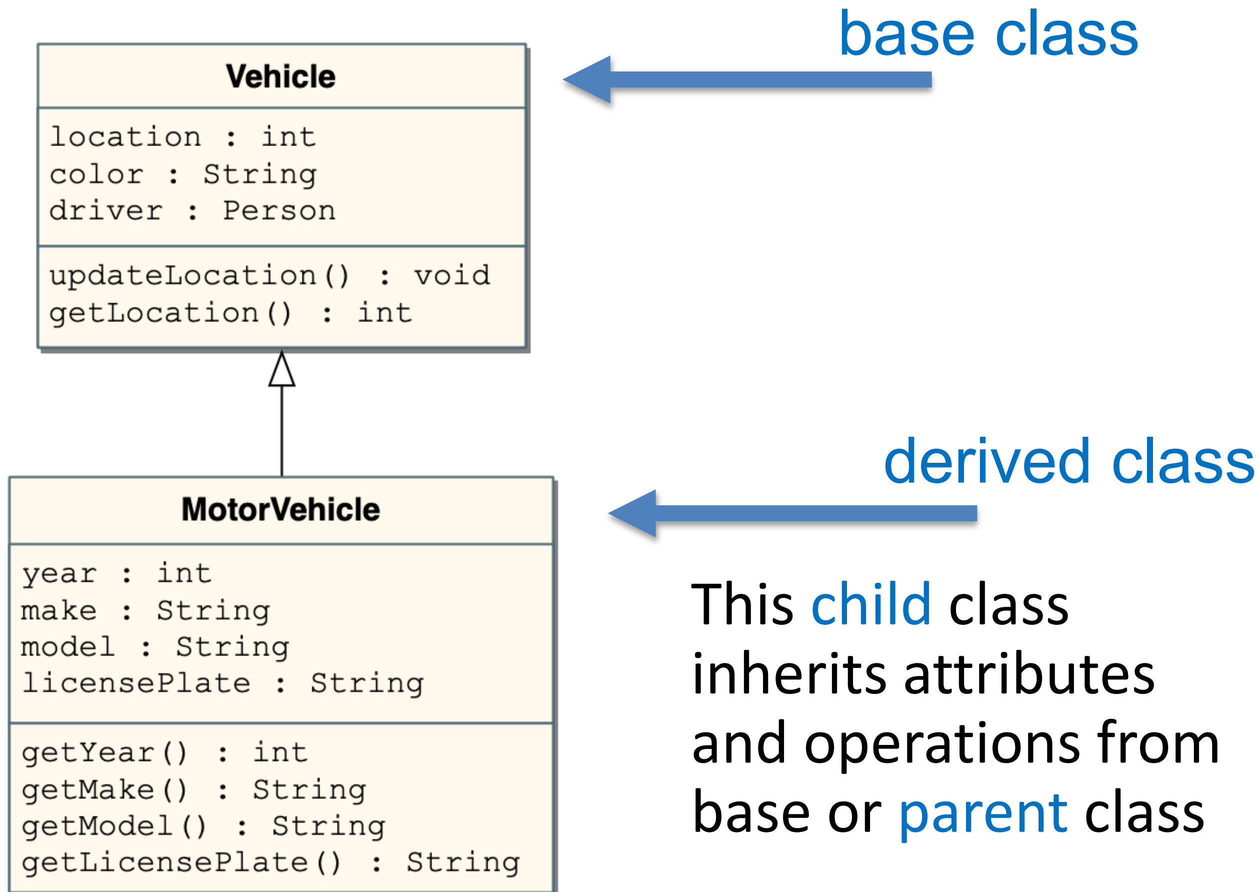
UML Class Diagrams

A UML class diagram for a `Vehicle` class:



Showing Inheritance

The arrow shows inheritance.

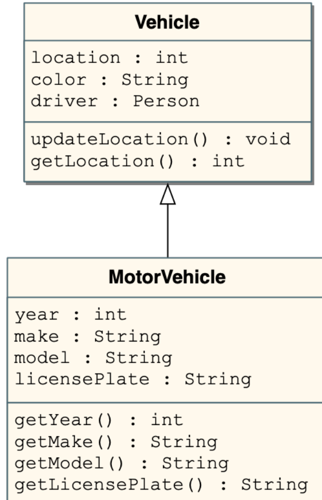


This child class inherits attributes and operations from base or parent class

References and Inheritance

An object reference can refer to an object of its class, or to an object of any class related to it by inheritance

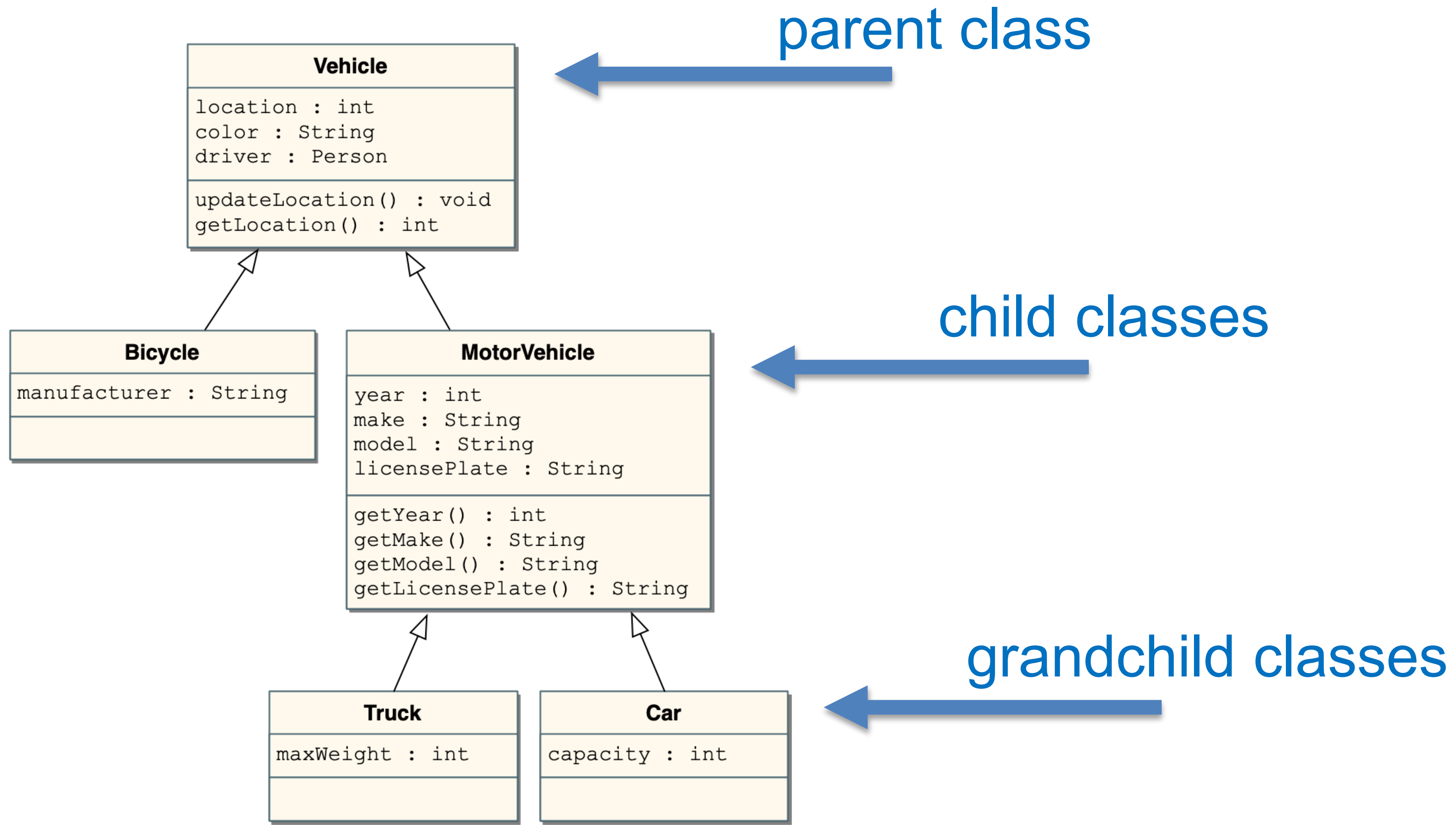
For example, if the `Vehicle` class is used to derive a class called `MotorVehicle`, then a `Vehicle` reference could be used to point to a `MotorVehicle` object




```
Vehicle vehicle;
vehicle = new MotorVehicle();

System.out.println(vehicle.getMake());
```

UML Class Diagrams

Class hierarchies can show multiple levels of inheritance.

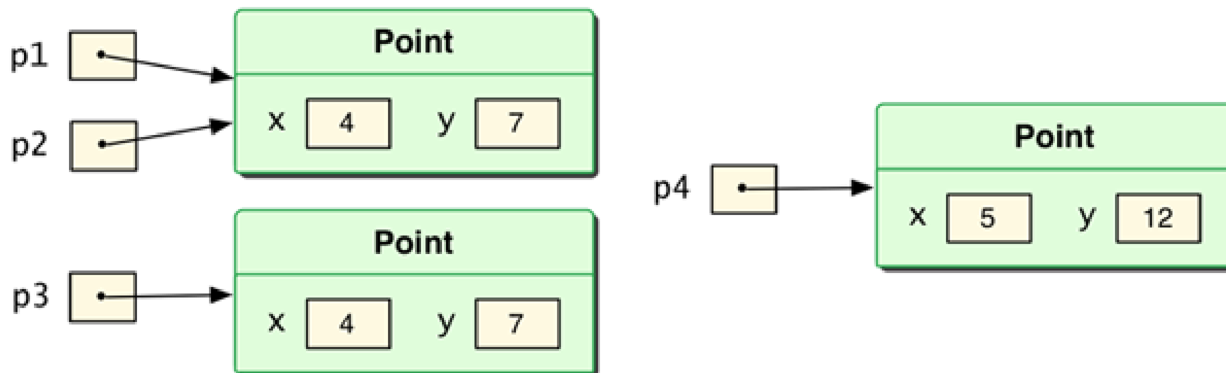


Rephactor

Object Equality

The following code creates three java.awt.Point objects and four Point reference variables:

```
Point p1 = new Point(4, 7);
Point p2 = p1;
Point p3 = new Point(4, 7);
Point p4 = new Point(5, 12);
```

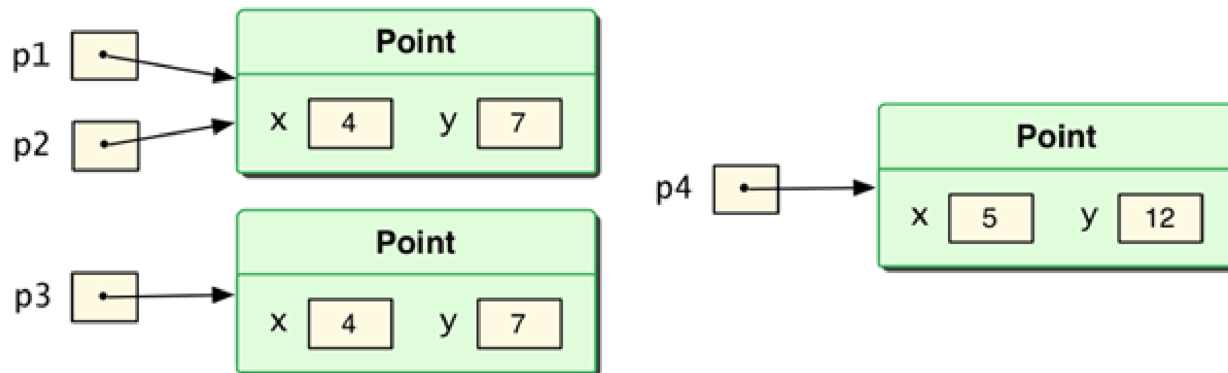


Object Equality

There are two types of object equality: reference equality and value equality

Two objects have reference equality if they point to the same object (like p1 and p2)

Two objects have value equality if they contain the same data (like p1 and p3)



Object Equality

Reference equality is determined by the == operator

Value equality is determined by the equals method

```
if (obj1 == obj2)
    System.out.println("Same object.");

if (obj1.equals(obj2))
    System.out.println("Equivalent content.");
```

The designer of a class gets to determine what it means for two objects to be equal

Object Equality

The **equals** method of the String class has been written to compare the characters in the string

Case matters ("Program" is not equal to "program")

But String also has an equalsIgnoreCase method

```
if (str1.equalsIgnoreCase(str2))
    System.out.println("Differ only by case.");
```

Object Equality

String literals with the same characters are mapped to the same object, which can lead to strange behavior

```
String s1 = new String("Java");
String s2 = "Java";
```

s1 == "Java"	false
s2 == "Java"	true

Both would be equal to "Java" according to the equals method

Bottom line: always use the equals method to compare strings

Object Equality

Following a null reference will cause an exception to be thrown

Use the == or != operator to avoid following a null reference

```
if (name != null)
    System.out.println(name.toUpperCase());
```

A null string reference is different than an empty string (""), which is a valid string of length 0

ENCAPSULATION

Encapsulation

Encapsulation is the principle that a class should protect its data from unnecessary access

This is about good design – creating software using elements that have limited, controlled interaction with other elements

If an object is encapsulated, problems are localized and easier to find and fix

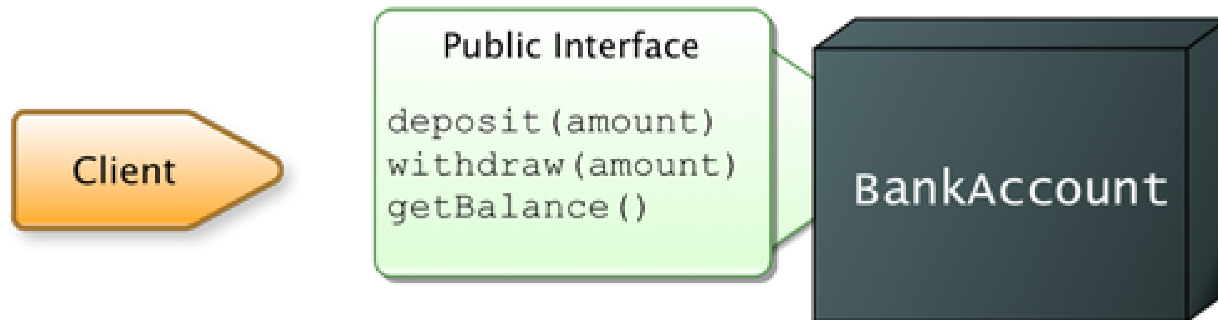
Code that interacts with an object is a client of that object

Encapsulation ensures that a client cannot "reach in" and change the values of instance variables directly

Encapsulation

From the client's point of view, an object should be a black box

The client doesn't need to know exactly what data is managed by an object or how it's organized



Instead, the client interacts with an object through its public interface – the set of methods a client can call

Encapsulation

A client shouldn't be able to set the balance of a bank account directly

In Java, encapsulation is accomplished using appropriate access modifiers, also called visibility modifiers

By declaring the account balance to be private, it can only be accessed and changed by methods in its class

	public	private
variable	Violates encapsulation	Enforces encapsulation
method	Part of the public interface	Supports other methods in the class

Encapsulation

An accessor method is a method that provides the client with the value of an instance variable

Some accessor methods take the form getX, where X is the attribute – this is also called a "getter" method

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

Encapsulation

Methods that modify instance data are mutator methods

They might be classic "setters"

```
public void setName(String newName)
{
    name = newName;
}
```

Unconditional mutators are almost as bad as public variables, but at least they are explicitly defined as such

The deposit and withdraw methods of a bank account are mutators because they change the balance

Some methods are both accessors and mutators

Example: Person

Explore **Example: Person** topic in Rephactor

Things to look for:

- **public class Person** – the declaration and "name" of the class
- **instance data or fields** – the "attributes"
- **getters and setters** – "operations" to get or set attributes, sometimes called **accessors**
- **other methods** – more "operations"

Method Anatomy

A method declaration defines the code that is executed when the method is called

A method must be declared inside a class

A method declaration consists of the method header followed by the method body

The method header includes modifiers, the return type, the method name, and the parameter list

The method body is enclosed in { } and contains the statements that will be executed each time the method is invoked

Method Anatomy

If a method doesn't return a value, it's return type is void

This method takes no parameters and returns no value:

```
public void printLyrics()
{
    System.out.println("Is this the real life?");
    System.out.println("Is this just fantasy?");
    System.out.println("Caught in a landslide,");
    System.out.println("No escape from reality.");
}
```

Method Anatomy

A return statement is used to return a value from the method

Its expression must be consistent with the return type

```
public String getGreeting()
{
    return "Hello! Glad you could join us!";
}
```

If a method doesn't return a value, the return statement is usually omitted

A method automatically returns when it reaches the end of the method body

Method Anatomy

A parameter is specified using a type and a name

The parameter serves as a temporary variable in the method and takes on the value passed to it

```
public static String getGreeting(String name)
{
    return "Hello, " + name + "! Glad you could join us!";
}
```

The parameters in the method header are formal parameters

The values passed in by the calling method are called actual parameters or arguments

Method Anatomy

If the method is called from the main method, it must also be declared as **static**

```
public static void main(String[] args)
{
    System.out.println(getGreeting("Ike"));
    System.out.println(getGreeting("Tina"));
}

public static String getGreeting(String name)
{
    return "Hello, " + name + "! Glad you could join us!";
}
```

```
Hello, Ike! Glad you could join us!
Hello, Tina! Glad you could join us!
```

Method Anatomy

Parameters and the return type can be any primitive or object type

```
public double circleArea(int radius)
{
    return Math.PI * radius * radius;
}
```

Method Anatomy

Multiple parameters are separated by commas

Each parameter has its own type

```
public int max(int num1, int num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}
```

When a return statement is executed, the method returns immediately

Method Anatomy

This method returns true if the first parameter is evenly divisible by the second

It also checks to see if the second parameter (the divisor) is zero

```
public boolean isDivisible(int num1, int num2)
{
    if (num2 == 0)
        return false;
    else
        return (num1 % num2 == 0);
}
```

Example: Bank Account

Let's examine a class that defines a bank account

Our simplified account will only manage an account number and the current balance

Explore the **Example: Bank Account** topic in Rephactor.