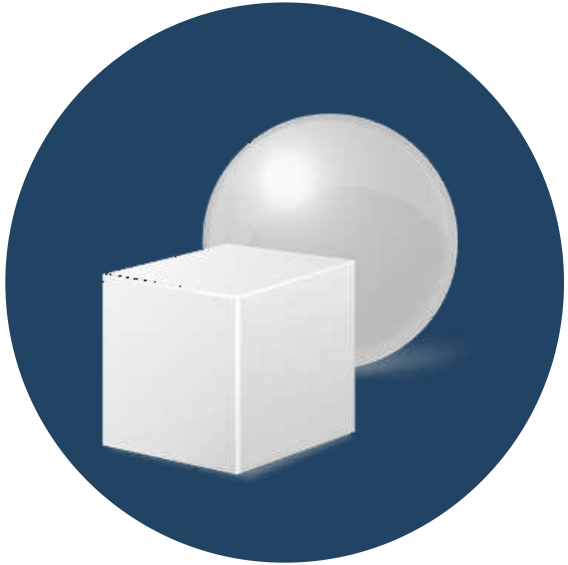

MY CODE DOESN'T WORK
LETS CHANGE NOTHING AND RUN IT AGAIN
quickmeme.com

Module 1-09

Objects and Classes
(Encapsulation)
(Static Members)
(Overloading)



Objects and Classes

Classes

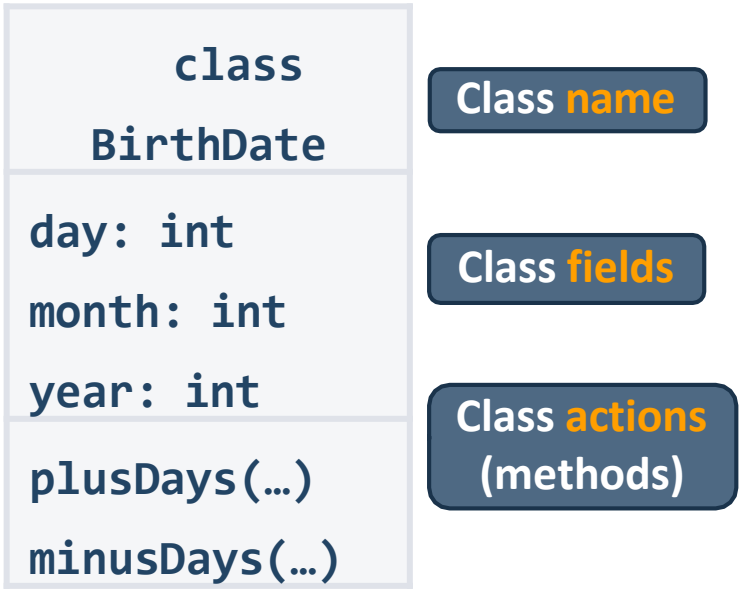
- In programming **classes** provide the structure for **objects**
  - Act as a **blueprint** for **objects** of the same type
- Classes define:
  - **Fields** (**private variables**), e.g. day, month, year
  - **Getters/Setters**, e.g. `getDay`, `setMonth`, `getYear`
  - Actions (**behavior**), e.g. `plusDays(count)`, `subtract(date)`
- Typically a class has multiple **instances** (objects)
  - Sample class: **`BirthDate`**
  - Sample objects: **`birthdayPeter`**, **`birthdayMaria`**

Objects - Instances of Classes

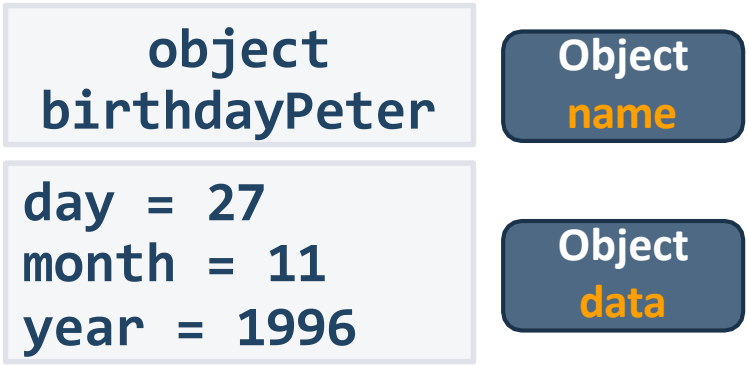
- Creating the object of a defined class is called **instantiation**
- The **instance** is the object itself, which is created runtime

Classes vs. Objects

- Classes provide structure for creating objects



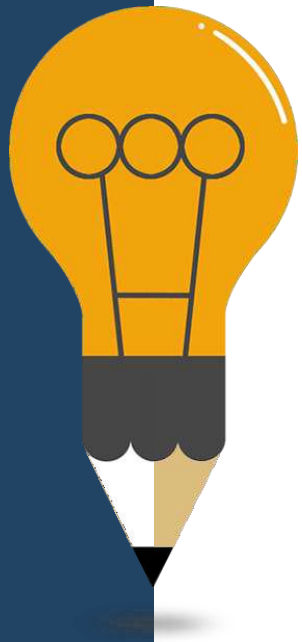
- An object is a single instance of a class

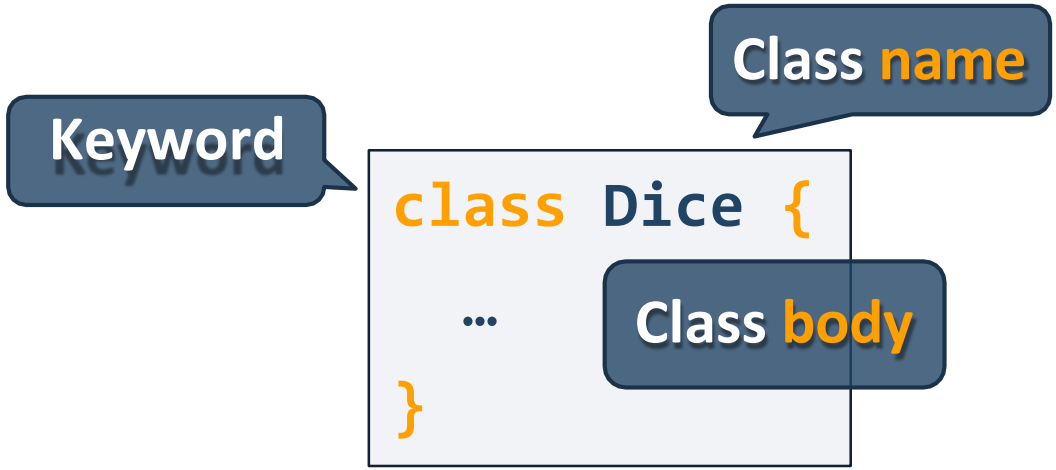


- # Defining Classes
  - ## Creating Custom Classes

Defining Simple Classes

- Specification of a given type of objects from the real-world
- **Classes** provide structure for describing and creating objects





Naming Classes

- Use **PascalCase** naming
- Use **descriptive** nouns
- Avoid abbreviations (except widely known, e.g. URL, HTTP, etc.)

```
class Dice { … }
class BankAccount { … }
class IntegerCalculator { … }
```

```
class TPMF { … }
class bankaccount { … }
class intcalc { … }
```

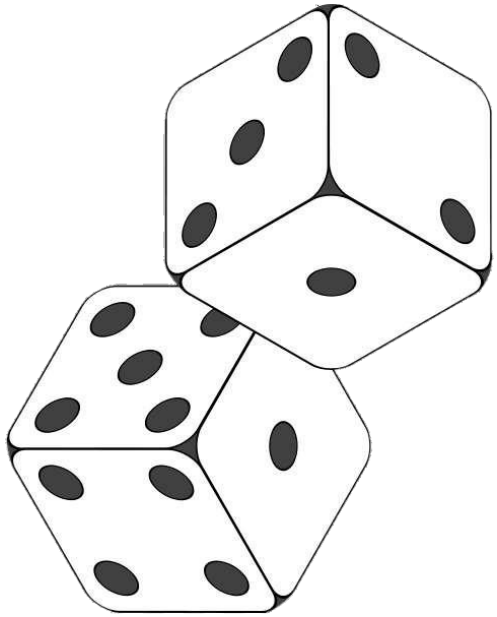
Class Members

- Class is made up of **state** and **behavior**
- Fields **store values**
- Methods **describe behaviour**

```
class Dice {
  private int sides;
  public void roll() { … }
}
```

Field

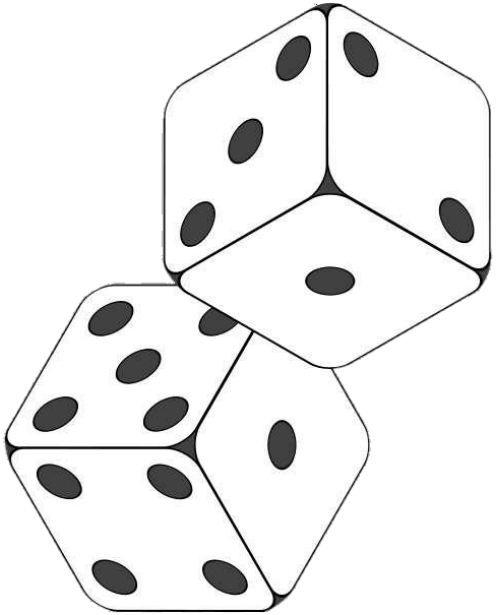
Method



Methods

- Store executable code (algorithm)

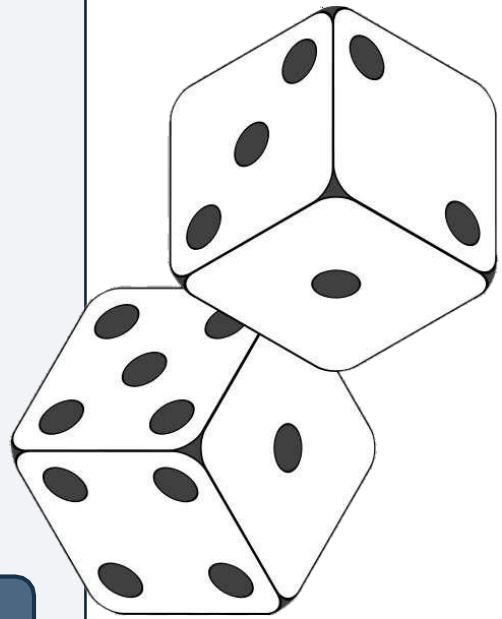
```
class Dice {
  public int sides;
  public int roll() {
    Random rnd = new Random();
    int sides = rnd.nextInt(this.sides + 1);
    return sides;
  }
}
```



Getters and Setters

```
class Dice {
  public int sides;

  public int getSides() { return this.sides; }
  public void setSides(int sides) {
    this.sides = sides;
  }
  }
```



Getters & Setters

Creating an Object

- A class can have many **instances** (objects)

```
class Program {
  public static void main(String[] args) {
    Dice diceD6 = new Dice();
    Dice diceD8 = new Dice();
  }
}
```

Use the **new** keyword

Variable stores a **reference**

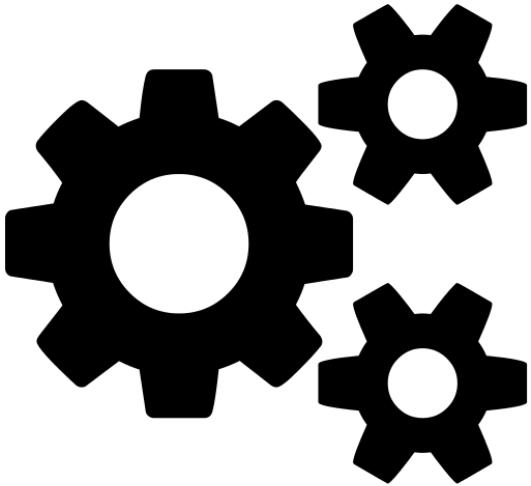
Constructors

- Special methods, executed during object creation

```
class Dice {
  public int sides;
  public Dice() {
    this.sides = 6;
  }
}
```

Constructor name is the same as the name of the class

Overloading default constructor



Constructors (2)

- You can have multiple constructors in the same class

```
class Dice {
   public int sides;
   public Dice() { }
   public Dice(int sides)
   {
      this.sides = sides;
   }
}
```

```
class StartUp {

public static void main(String[] args)

   { Dice dice1 = new Dice();

   Dice dice2 = new Dice(7);

   }
}
```

Summary

- Classes define templates for object
  - **Fields**
  - **Constructors**
  - **Methods**
- Objects
  - Hold a set of **named values**
  - **Instance** of a class

Intro to Classes

(Second Deck)

Module 1 - 09

Objectives

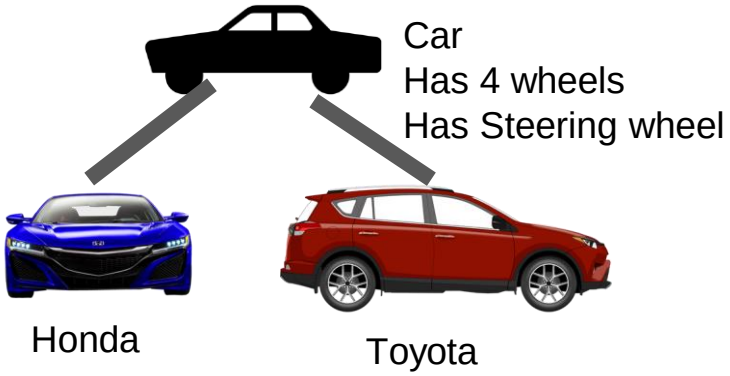
1. 3 Fundamental Concepts of OOP
2. Encapsulation
    - Loose Coupling
    - Access Modifiers
3. Defining Classes & Packages
4. Creating a Class
5. Class Members and this
    - Member Variables
    - Properties (Getters and Setters)
    - Derived Properties
    - Static and final variables
6. Member Functions
    - Methods
    - Constructors
7. Overloading
    - Method Overloading
    - Constructor Overloading
8. Static Methods

3 Fundamental Principles of Object Oriented Programming (OOP)

Encapsulation - the concept of hiding values or state of data within a class, limiting the points of access.

The key starts the car. The Ignition system is hidden from the user turning the key to start the car.

Inheritance - the practice of creating a hierarchy for classes in which descendants obtain the attributes and behaviors from other classes classes.



Polymorphism - the ability for our code to take on different forms. In other words, we have the ability to treat classes generically and get specific results.

Benefits of Object Oriented Programming (OOP)

- Benefits of OOP are:
  - A natural way of expressing real-world objects in code
  - Modular and reliable, allowing changes to be made in one part of the code without affecting another
  - Discrete units of reusable code
  - Units of code can communicate with each other by sending and receiving messages and processing data
- We will be talking about these more over the next week and a half

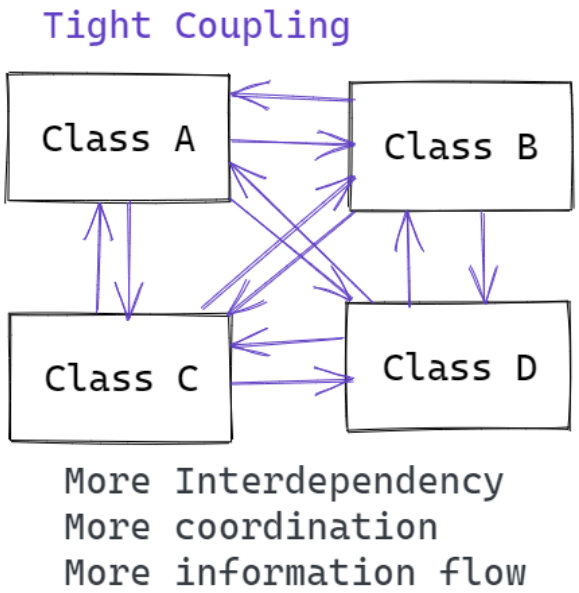
Encapsulation

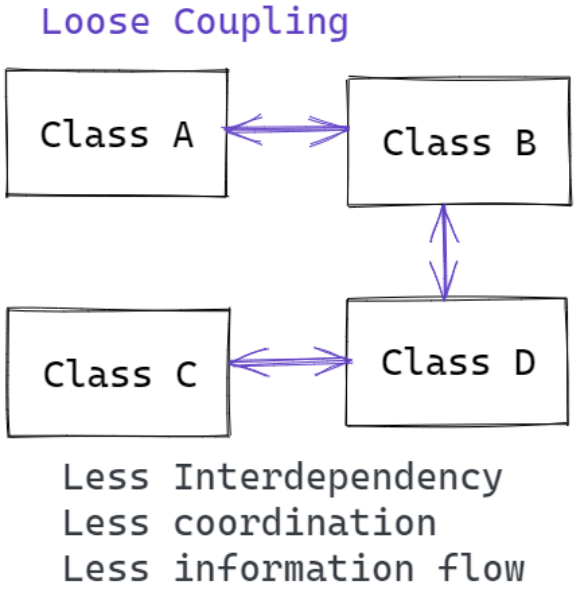
- the packaging of data and functions into a single component
- hiding the implementation details of a class to prevent other parties from setting the data to an invalid or inconsistent state and also to reduce **coupling**.
- Enables classes to be highly cohesive, meaning to have clear defined purpose
    - Highly cohesive classes have a clearly defined relationship with other classes
- Implemented by using *access modifiers* that let the compiler know which data members and methods can be accessed and modified by others.

Loose vs Tight Coupling

Coupling

Coupling refers to the degree of direct knowledge each component of a system or application has of elements that it needs to use. ***Loose Coupling*** is an approach to connect components that depend on each other with the least knowledge possible. Encapsulation promotes Loose Coupling.





Tightly Coupled

Loosely Coupled

Access Modifiers

- Visibility of classes, member methods, and variables
- Defines who can use a class, method, or variable.

Access Modifier	Description
public	Accessible to anyone who can use our class
private	Accessible only to code within the same class

Variables in our class should always be private. If the others need access to them, then they should be given access via public *getters* and *setters*.

Methods should be public only if they are meant for use by the users of our class

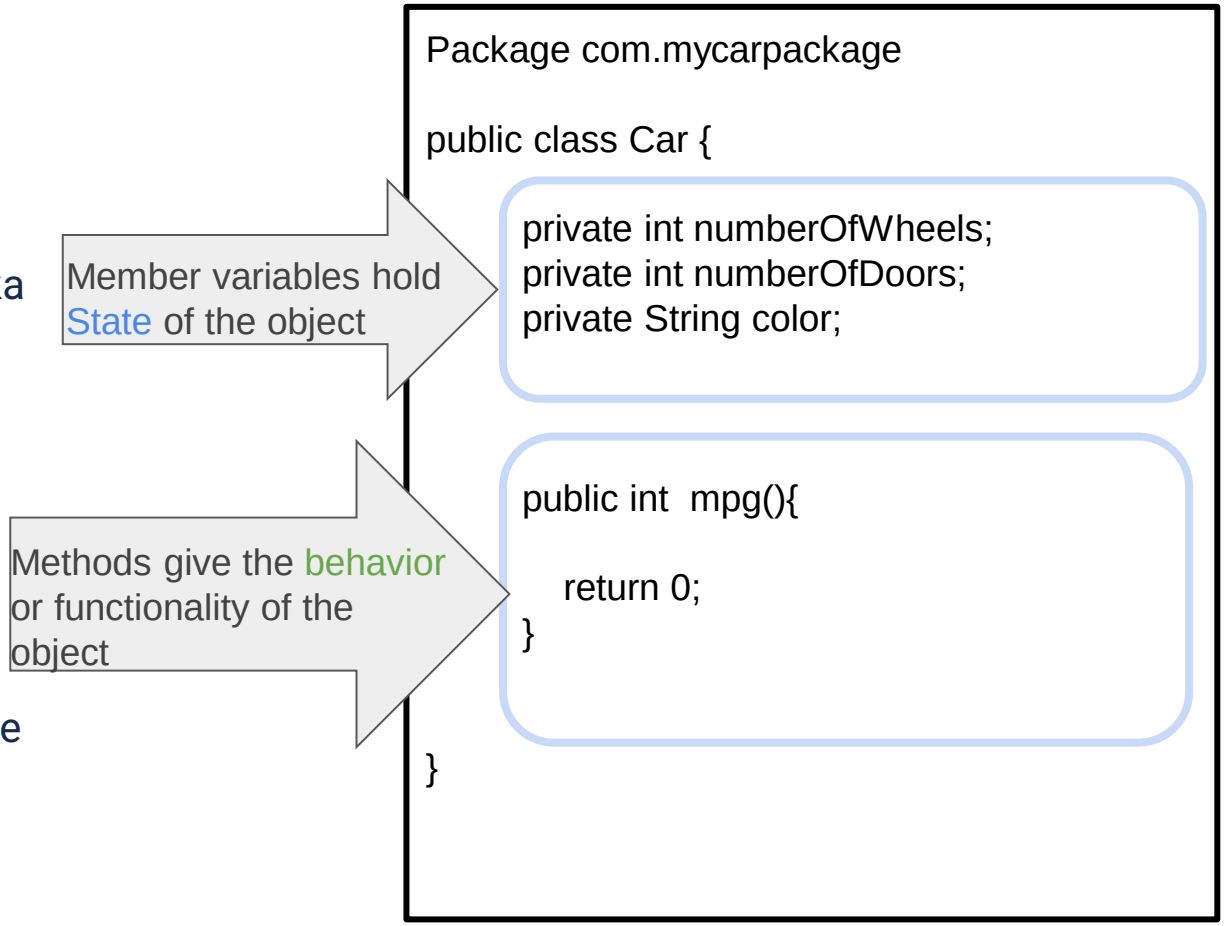
Why Limit access to parts of a class?

1. Makes code extendedable
2. Makes code maintainable
3. Promotes loose coupling

Classes

A class is a blueprint or model that defines state with fields (aka variables) and behavior with methods. We create new instances of a class that follow the blueprint but may have different property values.

All Reference Types are defined by a Class, and all Classes define a Data Type.



Class Naming

- Use *nouns or noun phrases*, not verbs
  - Short phrase that defines a thing
- Try to use the *singular form* of a class name
  - Vehicle not Vehicles
  - Car not Cars
- Class name must match the file name
  - Inside of Car.java file you have the class Car
- Follow Pascal Casing
  - 1st letter of every word is capitalized
    1. Ex. `userAccount` is in camel case and `UserAccount` is in Pascal case.

Member Variables

Member Variables also known as Instance Variables hold the data. Member variables create object state.

Access Modifiers define class variables that represent its properties. They control visibility to methods and properties to the rest of your program.

`public` member variables are **bad** because they

a. allow direct access to object internals
b. they don't allow the object to verify or modify data before it gets into the object state.

Getters and Setters should be the only way to access member variables from outside the class

Getter and Setter methods should always begin with the "get" or "set" prefix, except for Getter methods that return a boolean, which should begin with the prefix "is".

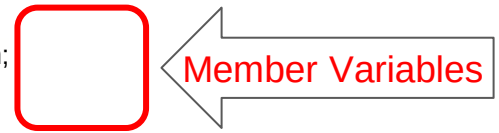
```
package com.techelevator;

public class Rectangle {

    private int length;
    private int width;

    public int getLength() {
        return length;
    }
    public void setLength(int length) {
        this.length = Math.abs(length);
    }
    public int getWidth() {
        return width;
    }
    public void setWidth(int width) {
        this.width = Math.abs(width);
    }

}
```



Access Modifiers

Access modifiers control who can access a variable or method.

public	Can be accessed by anyone
private	Can only be accessed from inside the Class.

Getters and Setters

Getters and Setters allow public access to a private member variable while still allowing the class to have full control of the variable.

```
private String taskName;
private boolean complete;

public String getTaskName() {
        return this.taskName;
}

public void setTaskName( String taskName ) {
        this.taskName = taskName ;
}

public boolean isComplete() {
        return this.complete;
}

public void setComplete( boolean complete ) {
        this.complete = complete;
}
```

member variables

Getter

Setter

Getter

Setter

this

This ***this*** keyword refers to the member variable specific to the instance of an object where the code is run.

```
public class Car {

            private String color;

            public String getColor {
                        return this.color;
            }

            public String setColor(String
color) {
                        this.color = color;
            }

}
```

```
Car blueCar = new Car();
blueCar.setColor( "Blue" );
```

this instance

```
Car redCar = new Car();
redCar.setColor( "Red" );
```

this instance

Derived Properties

A derived property is a **getter** that, instead of returning a member variable, returns a *calculation* taken from member variables. If we have `firstName` and `lastName`, we don't need to also store `fullName`, we can derive it from what we already have.

```
package com.techelevator;


public class Rectangle {


    private int length;
    private int width;


    public int getLength() {
        return length;
    }
    public void setLength(int length) {
        this.length = Math.abs(length);
    }
    public int getWidth() {
        return width;
    }
    public void setWidth(int width) {
        this.width = Math.abs(width);
    }


    public int getArea() {
        return this.length * this.width;
    }

}
```

final

Sometimes we have constant values, or values that can not be changed, that we want to use in our code without duplicating the value all over the place. Many languages have the concept of constant and global constant variables, however, Java does not, but we can mimic one using the final keyword.

The final keyword allows the variable to be assigned once, but after set, it cannot be reassigned.

```
1) private final int x = 10;
2) x = 15;  ← not allowed, since x is final and has a value
```

```
1) private final int y;
2) y = 20;   ← allowed, y was declared as final, but not yet assigned
3) y = 30;   ← not allowed, since y is final and has a value.
```

static variables

Static members belong to the class. ***Instance members belong to an instance of the class***. Static methods can be invoked without creating an instance of the class. Static variables and methods cannot be accessed with the **this** keyword, since they are not part of the object.

```
1) private static int x = 10;
2) int y = x + 5;
```

```
int y = this.x + 5;  ← the keyword this cannot be used with static
```

Since static variables are shared by all instances of a *class* and not the individual *object*. Changes to the value from one object can be seen from all objects of that type.

Methods

A **function** or method, is like a mathematical function (e.g. `f(n) = n^2`). Methods can have multiple parameters but can only return one value (for now).

Public methods define object behaviors.

Public and Private Methods help by

- making the code base manageable with smaller chunks
- reducing code into small units of work, making debugging simpler
- and introducing reuse.

Method Signature

- All methods have a name
  - usually a verb or verb phrase that describes the action
- All methods have a return type
  - what the method will return
  - could be anything: Rectangle, boolean, ***void (returns nothing)***
- Methods can be parameterless or can include parameters (or inputs).

public <returnType> name (<List of Arguments>)

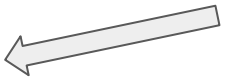
```
public boolean isLargerThan(int length, int width) {
    return (this.length * this.width) > (length * width);
}
```

Function Overloads

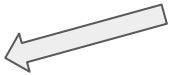
Overloaded methods are methods with the *same name and return type*, and a *different set of parameters*. Java uses the correct overload based on the parameters sent to it.

```
public boolean isLargerThan( Rectangle other) {
        return this.area > other.getArea();
}

public boolean isLargerThan( int width, int height ) {
        return this.area > width * height;
}
```



```
ourRectangle.isLargerThan( otherRectangle );
```



```
ourRectangle.isLargerThan( 10, 12 );
```

Constructors

A **constructor** is a special method that runs every time a new object is instantiated. It allows for the object to be initialized with a starting state.

- The name must exactly match the Class name
- It has no return type
- Can have arguments that set the value of member variables, like a Setter.
- Can be Overridden to provide multiple ways to instantiate the object
- If no constructor is present, then Java provides a ***default constructor***, which has no arguments.
  - if any constructor is present then the default constructor is not provided, and if a no-argument overload of the constructor is needed, then it must be explicitly created.

```
Scanner input = new Scanner( System.in );
```
← System.in is being passed to the constructor of Scanner.

```
myStr = new String();
```
String ← The () in an object instantiation calls the constructor. Here the no-argument constructor is being called to create a String object with no starting value;

`public class Car {`
 	
`private String color;`
	Default Constructor:
	`Car myCar = new Car();`

`public class Car {`
 	
`private String color;`
 	
`public Car( String color ){`
`this.color = color;`
`}`
	Constructor with argument. The argument value must be passed to instantiate the Car object.
	`Car myCar = new Car( "red" );`

`public class Car {`
 	
`private String color;`
 	
`public Car() { }`
 	
`public Car( String color ){`
`this.color = color;`
`}`
	No-Argument Constructor with a constructor Overload that allows a value to set the starting state at instantiation.
	`Car myCar = new Car();`
	`Car myCar = new Car( "red" );`

Function Overriding

Some methods will be “inherit” from other classes. For example, all class “inherit” the methods toString() and equals()

If we want to provide our own functionality for these methods we can create an “**Override**” which will hide the original methods functionality and provide our own.

To override a method, we must provide an **identical method signature as the one being overriden. Though not required, It also a good practice to include the @Override annotation for readability.**

```
@Override
public String toString() {
        return “our string representation of
our object”;
}
```

```
@Override
public boolean equals(Object obj) {
        return if the value of obj is equal to the
        value of this object

}
```

static methods

If we define a method static, our class does not need to be instantiated to use it. Instead it can be accessed from the Class itself, instead of the object. *Static methods can only access other static methods or variables.*

```
public class Rectangle() {

        private static int length;
        private static int width;

        public static getArea() {
                    return length * width;
        }
}
```

```
private static void main(String[] args)
```

In the class using the static method:

```
        Rectangle.getArea();

        Rectangle rect = new Rectangle();
        rect.getArea();
```

This seems easier, why not make everything static?

Examples

Math.abs()
Math.random()

String.join()
String.valueOf()

Double.parseDouble()
Integer.parseInt()

Intro to Classes

(Third Deck)

Objects and Classes
Encapsulation
Static Members

Objectives

- **Define encapsulation**, give a good example of it, how it is implemented, and describe why it is used
- Define **"loosely coupled"** and explain the characteristics of a loosely coupled system
- Describe **constant variables**, how to create them, and their use
- Define and use **static methods** and be able to describe what they are for

Principles of Object-Oriented Programming (OOP) it's as easy as PIE!

- **Encapsulation -** the concept of hiding values or state of data within a class, limiting the points of access
- **Inheritance -** the practice of creating a hierarchy for classes in which descendants obtain the attributes and behaviors from other classes
- **Polymorphism -** the ability for our code to take on different forms through the use of overloading and overriding
- **(Abstraction) –** extension of encapsulation. We can't build a car from scratch, but we know how to use (drive) it.

Encapsulation!

Encapsulation

Encapsulation & Data Hiding

- **Encapsulation** is the process of combining related data members and methods into a single unit.
  - In Java, encapsulation and data hiding are achieved by putting all related data members and methods in a class.

- **Data hiding** is the process of obscuring the internal representation of an object to the outside world.
  - In Java, data hiding is achieved by setting all members to private and providing getters and setters for said members.

Encapsulation

Rule: instance variables (properties, data members) are private and methods are public

```
public class Car {
        private int year;

        public void setYear(int year) {
            this.year = year;
        }
        public int getYear() {
            return this.year;
        }
}
```

private can only be accessed or used inside the Car class

public means this method can be called outside of this class

Goal of Encapsulation

- Makes code extendable
- Made code maintainable
- Promotes “loose coupling”
  - Each of its components has or makes use of little or no knowledge of the definitions of other separate components

Static

Definition of Static in Java

If a method or data member is marked as static, it means **there is exactly one** copy of the method, or one copy of the data member shared across all objects of the class.

One way to think about this, is that the static member is a unique property of the “blueprint” that is the same for all objects created from that blueprint.

FordCar class might have a static data member logo. All FordCar objects will share the same static data member.

The non-static methods and data members we have defined so far are often called Instance members or Instance methods.

Static Members: Declaration

Static members and methods are declared by adding the keyword static.

```
public class Car {
		public static String carBrand = "Ford";

		public static void honkHorn() {
				System.out.println("beeep?");
		}
		...
}
```

Static: Calling

Assuming we have the static member declarations from the previous slide, this is how you call them from a different class. Note that we should use the class name (Car) as opposed to the name of an instance of a car (thisCar).

```
public class Garage {

        public static void main(String args[]) {

                System.out.println(Car.carBrand); // Correct way to refer to a static member.
                Car.honkHorn(); // Correct call to a static method.

                Car thisCar = new Car("Red", 2);
                System.out.println(thisCar.brand); // Not a typical way to call a static member.
                thisCar.honkHorn() // Not a typical way to call a static member

        }
}
```

Static: Assignment

Public Static data members can be reassigned to new values.

```
public class Garage {

        public static void main(String args[]) {
        Car.carBrand = “GM”;
        }
}
```

Static: Constants

Constants are variables that cannot change. The closest thing to a constant in Java is declaring a data member with **static final**.

```
public class Car {
		public static final String CAR_BRAND = "Ford";
...
}
```

Attempts to change the value of this data member will result in an error. This, for example is invalid:

```
public class CarDealership {

		public static void main(String args[]) {

				Car.CAR_BRAND = "GM";
		}
}
```

Static: Rules

There are some rules to observe when using static methods or data members:

- **Static** variables can be accessed by **Instance** methods.
- **Static** methods can be accessed by **Instance** methods.

The opposite of the above is not true:

- **Static** methods cannot access **Instance** data members.
- **Static** methods cannot call **Instance** methods.

Static: Rules

```
String someInstanceVariable;

public static void someStaticMethod() {
        System.out.printlnString (someInstanceVariable);
        someInstanceMethod();
}

public void someInstanceMethod() {

}
```

This is an instance (non-static data member)

We are inside a static method, but we are referencing an instance member, which is not allowed

We are inside a static method, but we are calling an instance method, which is not allowed.

You have encountered this issue before - recall that any method directly called by public static void main had to also be a static.

Objectives - Review

- **Define encapsulation**, give a good example of it, how it is implemented, and describe why it is used

Encapsulation in Java

← Prev Next →

Encapsulation in Java is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.

We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

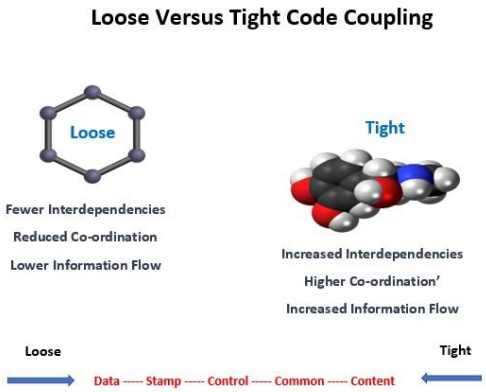
The **Java Bean** class is the example of a fully encapsulated class.

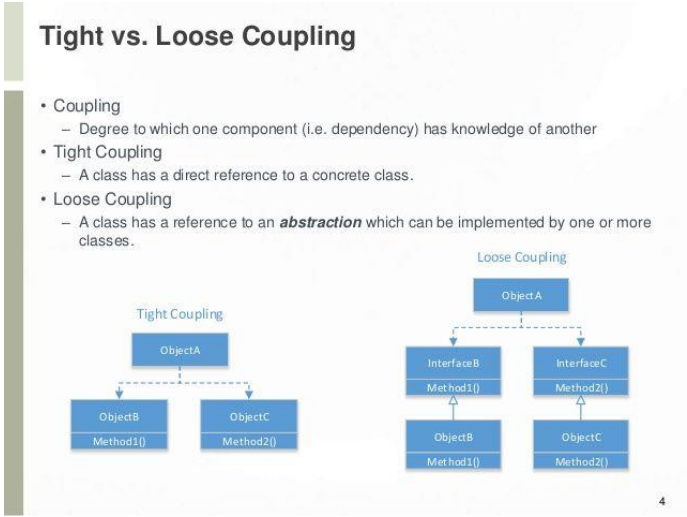
Capsule

https://www.javatpoint.com/encapsulation

Objectives

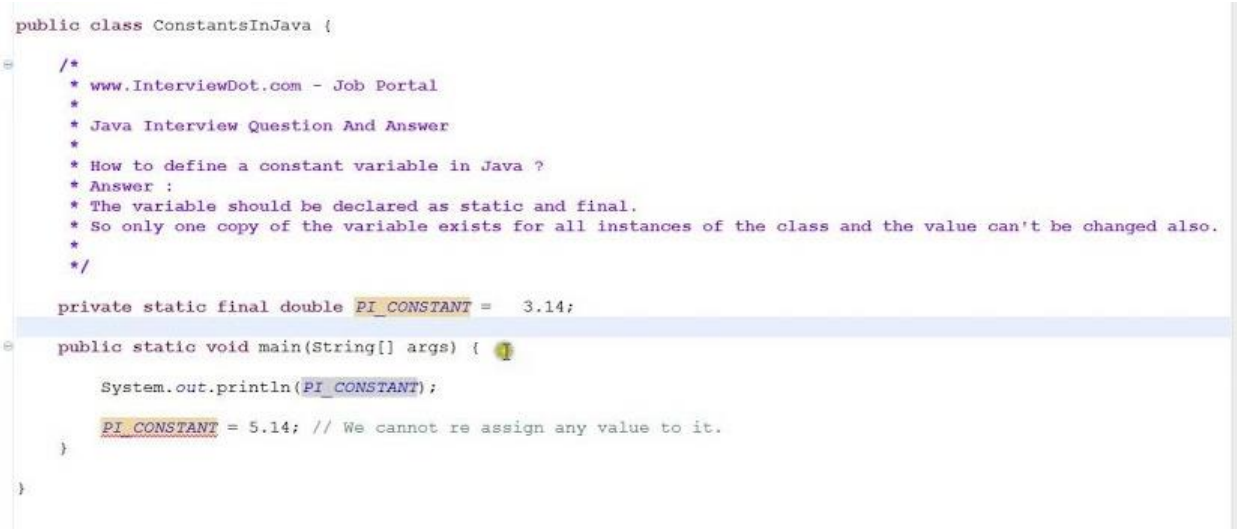
- **Define encapsulation**, give a good example of it, how it is implemented, and describe why it is used
- Define **"loosely coupled"** and explain the characteristics of a loosely coupled system





Objectives

- **Define encapsulation**, give a good example of it, how it is implemented, and describe why it is used
- Define **"loosely coupled"** and explain the characteristics of a loosely coupled system
- Describe **constant variables**, how to create them, and their use



```
public class ConstantsInJava {

    /*
     * www.InterviewDot.com - Job Portal
     *
     * Java Interview Question And Answer
     *
     * How to define a constant variable in Java ?
     * Answer :
     * The variable should be declared as static and final.
     * So only one copy of the variable exists for all instances of the class and the value can't be changed also.
     *
     */

    private static final double PI_CONSTANT =   3.14;

    public static void main(String[] args) {

        System.out.println(PI_CONSTANT);

        PI_CONSTANT = 5.14; // We cannot re assign any value to it.
    }

}
```

Objectives

- **Define encapsulation**, give a good example of it, how it is implemented, and describe why it is used
- Define **"loosely coupled"** and explain the characteristics of a loosely coupled system
- Describe **constant variables**, how to create them, and their use
- Define and use **static methods** and be able to describe what they are for