# OOP Concepts

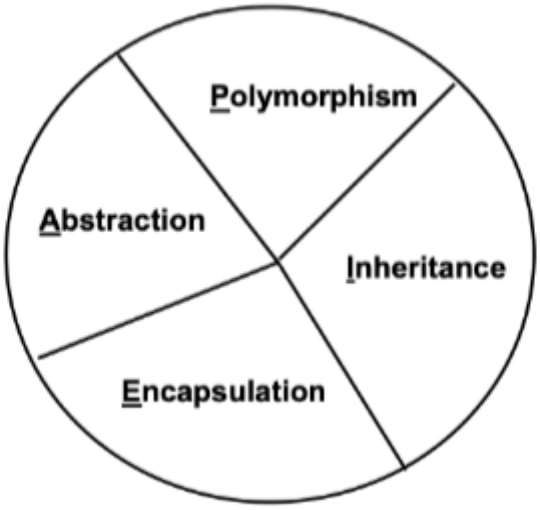
---

## Classes and Objects

- Blueprint for object
- Obj var's explain internal state of obj
- Obj methods specify action of obj

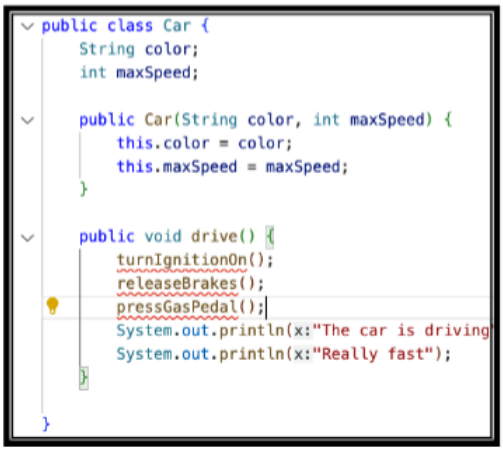
## The OOP Pie

- Four distinct OOP Concepts make “A PIE”




## Abstraction

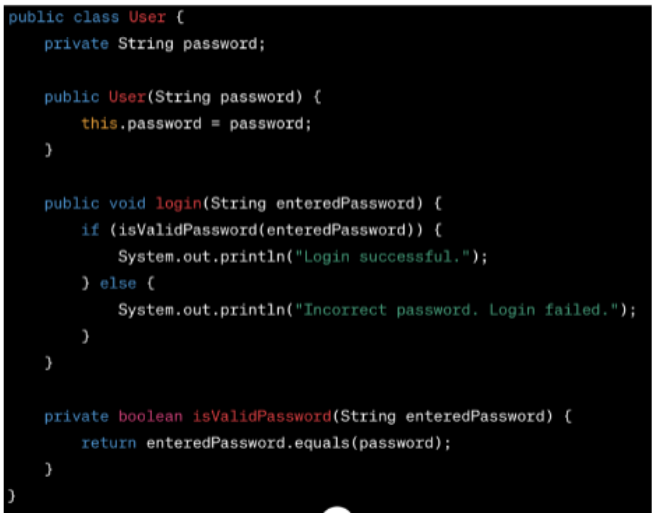
- Only show **necessary details** of object

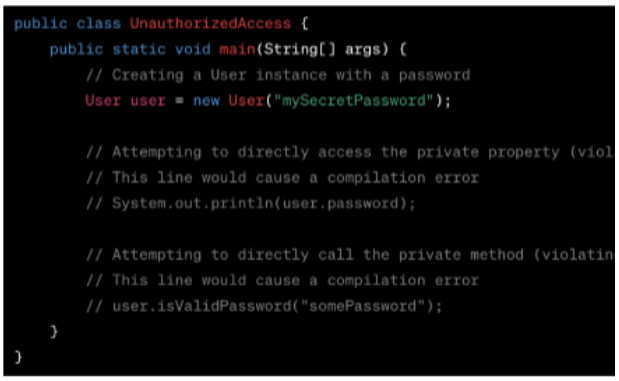




# Encapsulation

- Attributes and methods that operate on the data into a single unit known as a class
- **Protects** sensitive data and restricts access to it
- **Controls** access into the object
- Promotes data hiding
- **Access Modifiers**
  - private, public, protected, and default are access modifiers
  - **Private** used for encapsulation
- **Private Fields**
  - Private fields not accessible outside the class.
- **Accessors**
  - Methods that provide controlled access to *private* fields
  - Also known as *getters* and *setters*



# Inheritance

- Children classes can **inherit** parent classes functions/methods
- Allows for **method/constructor overriding**
  - Can use `super()` to call constructor of the superclass (parent class)
- For encapsulation, we can use `protected` for our variables if you want to share the parent variables to the children and no one else.
- **Abstract Classes**
  - Classes which declare but don't define behavior
- **Abstract Methods**
  - Methods which don't contain implementations

```java
public abstract class AbstractAnimal {

  public abstract void eat();

  public abstract void makeNoise();
}
```

```java
public class Dog extends AbstractAnimal {

  public void eat() {
    System.out.println(x:" i eat kibble!");
  }

  public void makeNoise() {
    System.out.println(x:" i bark!");
  }
}
```

```java
Run | Debug
public static void main(String[] args) {
  Dog buddy = new Dog();
  buddy.eat();
  buddy.makeNoise();
}
```

```
i eat kibble!
i bark!
```

- **Specified with the keyword "extends" :**

```java
public class Vehicle {

  private int weight;
  private double purchasePrice;
  //... other Vehicle data here

  public Vehicle ()
  { ... }

  public void turn (int direction)
  { ... }

  // ... other Vehicle methods here
}
```

```java
public class Truck extends Vehicle {
  private int freightCapacity;
  //... other Truck data here

  public Truck ()
  { ... }

  // ... Truck-specific methods here
}
```

- **Note: a Truck "is-a" Vehicle**
- **Only a single "extends" allowed (no "multiple inheritance")**
- **Absence of any "extends" clause implies "extends Object"**

## Polymorphism

- Types of Polymorphism
  - Compile-time (**Static**) Polymorphism
    - Method Overloading
  - Run-time (**Dynamic**) Polymorphism
    - Method Overriding
- Upcasting

- Casting a child object to its parent type
- Allows child obj to be treated as parent obj (generic)

```
Cat kitty = new Cat();
Animal kit = kitty;
kit.useLitterBox();
```

```
// }
The method useLitterBox() is undefined for the type Animal Java(67108964)
View Problem (⌥F8)  Quick Fix... (⌘.)
Animal
kitty.useLitterBox();
```

- Downcasting
  - Casting a parent object to its child type
  - Allows accessing child class-specific methods and properties

```
Animal animal = new Animal();
Cat kitty = (Cat) animal;
kitty.useLitterBox();
```

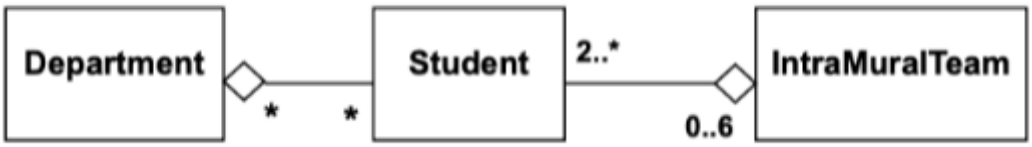
# Java Packages

- Used to group together classes belonging to the same category or providing similar functionality

# Associations

## Aggregation

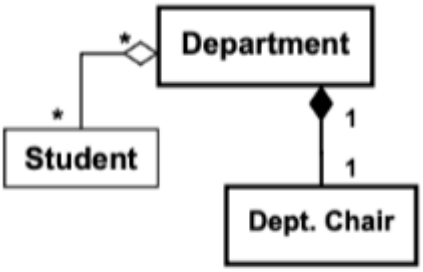
- Represents "**has-a**" or "**is-part-of**"



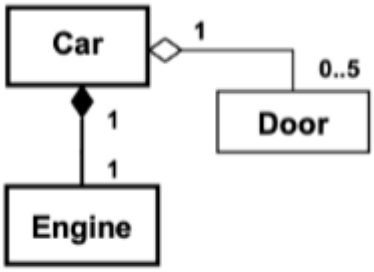

## Composition

- **Exclusive** Ownership
  - Without whole, part can't exist

- **Required** Ownership
  - Without part, the whole can't exist



Exclusive ownership



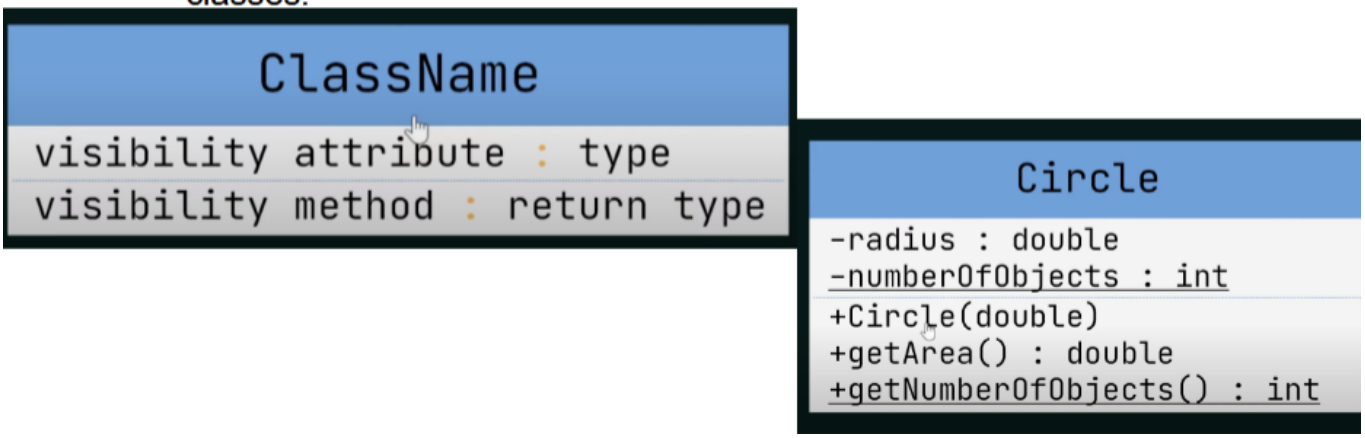
Required ownership

# UML

## UML – Class Diagrams

**Class Diagrams:** Represent the static structure of a system

- including classes, attributes, methods, and relationships between classes.

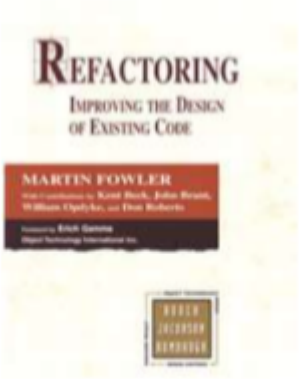



# Code Smells

---

## Martin Fowler's Smells

- **Duplicated Code**
- **Long Method**
- **Large Class**
- **Long Parameter List**
- **Divergent Change**
- **Shotgun Surgery**
- **Feature Envy**
- **Data Clumps**
- **Primitive Obsession**
- Switch Obsession
- Switch Statements
- Parallel Inheritance Hierarchies
- **Lazy Class**
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- **Inappropriate Intimacy**
- Alternative Classes with Different Interfaces
- Incomplete Library Class
- **Data Class**
- Refused Bequest
- **Comments**




# More Code Smells

- Hard Coding
  - `Aliens[] a = new Aliens[3];`
- Magic Numbers
  - `double circ = 6.28*r;`
- Programming by Permutation
  - making small changes and testing
- Cargo Cult Programming
  - e.g. setters and getters without good reason *(Pacific Islands, WWII)*
- Premature Optimization
  - Typically 3% needs optimization
- Not Invented Here Syndrome
  - Don't reinvent the wheel
  - *The opposite is also a smell*
- Error Hiding
  - Should we catch and handle exceptions?
- Coding by Exception
  - Adding new handling for every recognized special case
- Tester-Driven-Development
  - Allowing bug reports to drive development of new features (putting out fires!)
- Busy Waiting
  - Continually checking for a condition
- Boat Anchor
  - Obsolete or useless code that continues to encumber the system.
- Action at a Distance
  - Code in one part affects completely different part
    - e.g. caused by globals
- Inappropriate Intimacy
  - Direct access of object internals
    - e.g. Much of the existing SubHunter code

# Detection

- SAST Testing
- Peer review

# Code Refactoring

# Refactoring

- Changing code to be cleaner
- We refactor to
  - Prevent design decay
  - simplify code
  - find bugs
  - better readability
  - code smells
- Refactor when changing existing code

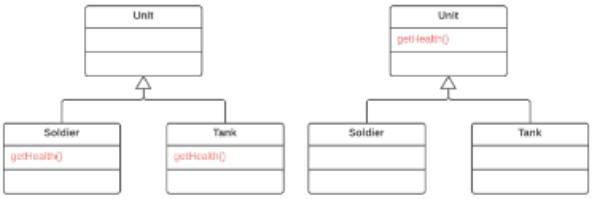
# Pull Up Method

## Problem

Your subclasses have methods that perform similar work.

## Solution

Make the methods identical and then move them to the relevant superclass.



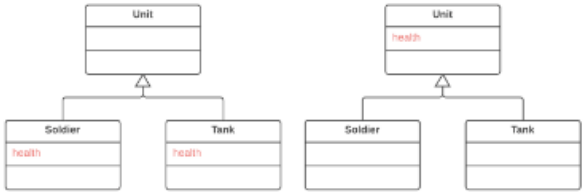

# Pull Up Field

## Problem

Two classes have the same field.

## Solution

Remove the field from subclasses and move it to the superclass.




# Pull Up Constructor Body

## Problem

Your subclasses have constructors with code that's mostly identical.

## Solution

Create a superclass constructor and move the code that's the same in the subclasses to it. Call the superclass constructor in the subclass constructors.

```
class Manager extends Employee {
  public Manager(String name, String id,
    this.name = name;
    this.id = id;
    this.grade = grade;
  }
  // ...
}
```

```
class Manager extends Employee {
  public Manager(String name, String id,
    super(name, id);
    this.grade = grade;
  }
  // ...
}
```

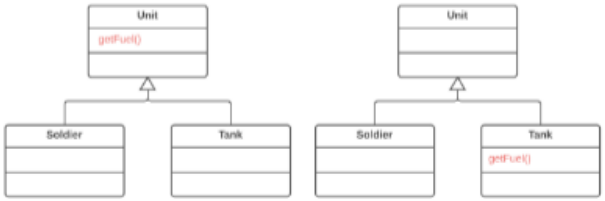
# Push Down Method

### Problem

Is behavior implemented in a superclass used by only one (or a few) subclasses?

### Solution

Move this behavior to the subclasses.



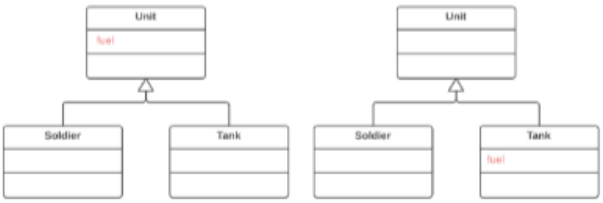

# Push Down Field

### Problem

Is a field used only in a few subclasses?

### Solution

Move the field to these subclasses.



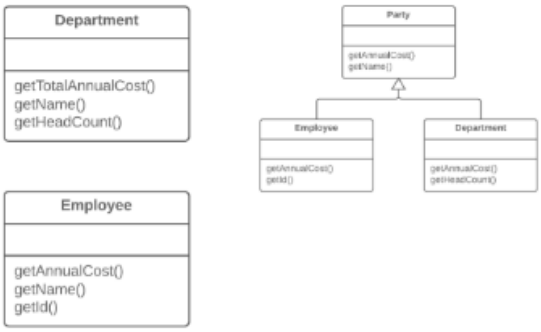

# Extract Superclass

### Problem

You have two classes with common fields and methods.

### Solution

Create a shared superclass for them and move all the identical fields and methods to it.



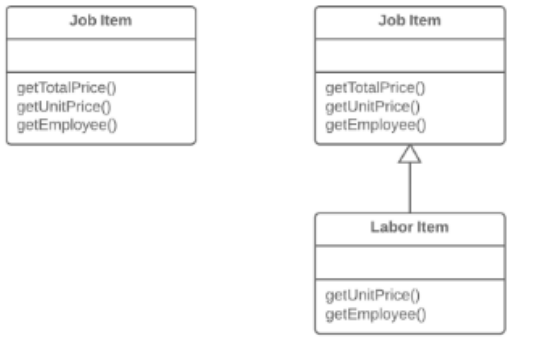

# Extract Subclass

### Problem

A class has features that are used only in certain cases.

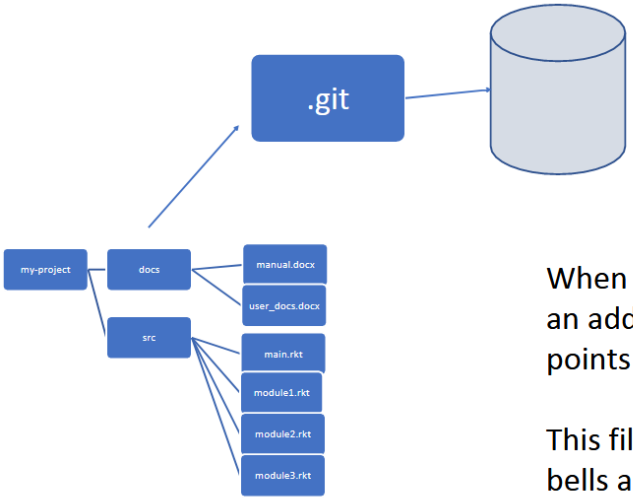
### Solution

Create a subclass and use it in these cases.




# Git

# Git Repo Files




When you have a git repository, you have an additional directory called .git, which points at a mini-filesystem.

This file system keeps all your data, plus the bells and whistles that git needs to do its job.

All this sits on your local machine.

# Setting up Global git

- `git config --global user.name "Your Username"`
- `git config --global user.email "Your email"`

# Best Practices

- Use descriptive commit messages
  - What commit does
  - Why is was made
- Commit Small Chunks of Code
  - Small, logical units of work
- Regularly Pull
  - Pull or fetch from remote repo to be up-to-date and not cause merge conflicts