OOP with Java

Objectives

- Understanding Object Oriented Programming
- OOP features
  - Abstraction
  - Encapsulation
  - Polymorphism
  - Inheritance
- Abstract classes & Interfaces
- Packages

Object-Oriented Programming

- Java is an object-oriented programming language
- As the term implies, an object is a fundamental entity in a Java program
- Objects can be used effectively to represent real-world entities
- For instance, an object might represent a particular employee in a company
- Each employee object handles the processing and data management related to that employee

Classes and Objects

- The ***class*** is the unit of programming
  - A class is a generic template for a set of objects with similar features
  - Defines the attributes and behavior of the objects
  - Each class definition (usually) in its own **`.java`** file
  - The file name must match the class name
- A class describes ***objects (instances)***
  - An instance is the specific concrete representation of a class.
  - It could be uniquely identified by its characteristics
  - These characteristics are:
    - ***Data fields*** for each object
    - ***Methods*** (operations) that do work on the objects

Abstraction

- Used to reveal the essential features of the object.
- In other words focus on the “big picture” and ignore specific details.
- Highlights the properties of an entity that we are more interested in and hides the others.
- In Java abstraction is achieved by
  - Abstract classes
  - Interfaces

Encapsulation

- Encapsulation and abstraction are commutual terms
- Refers to wrapping the data and functions that operate on the data together as a capsule
- Enhances security by restricting the access to data or member functions
- Encapsulation also increases modularity.
- Data fields are mostly private
- Constructors and accessor methods are public

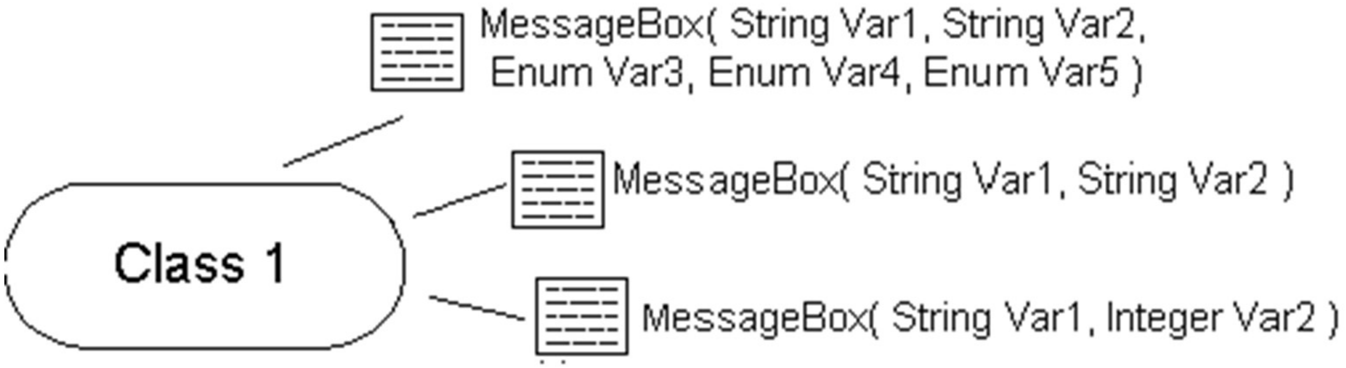
Polymorphism

- The ability of behavior to vary based on the conditions in which the behavior is invoked
- We define more than one function with the same name which varies with the parameters
- There are two type of polymorphism
  - Compile time polymorphism (overloading)
  - Runtime polymorphism (overriding)

Overloading

- Two or more methods with the same name but different signatures
- When a method is called it is associated to the definition with best matching signature
- If the message and the method have a different number of parameters, no match is possible.
- If the message and the method have exactly the same types of parameters, that is the best possible match.
- Messages with specific actual parameter types can invoke methods with more general formal parameter types
  - For example if the formal parameter type is Object, an actual parameter of type String is acceptable
  - If the formal parameter is type double, an actual parameter of type int can be used
- If there is no clear best match, Java reports a syntax error.

Overloading - Example



MessageBox(String Var1, String Var2, Enum Var3, Enum Var4, Enum Var5)

MessageBox(String Var1, String Var2)

MessageBox(String Var1, Integer Var2)

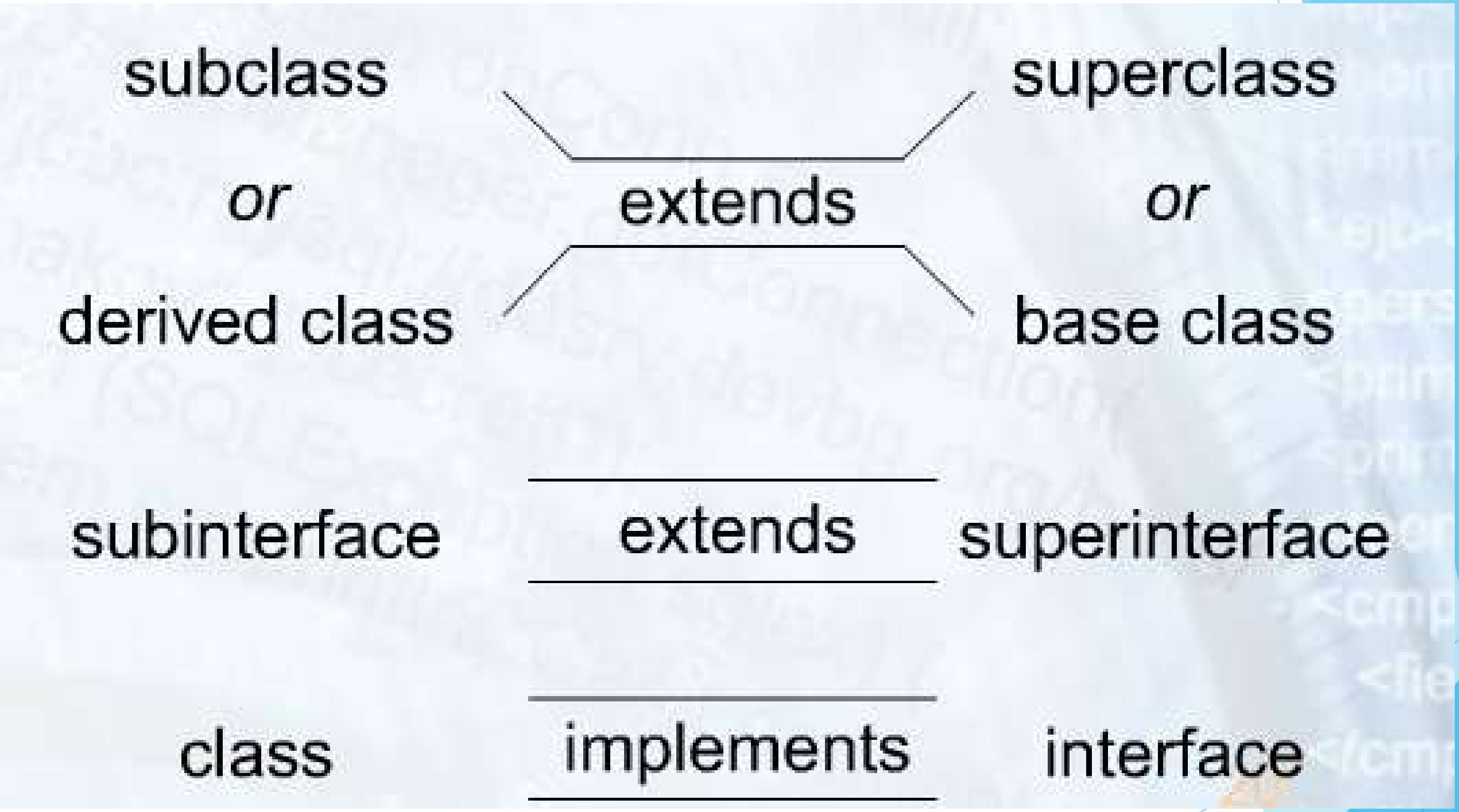
Overriding

- Occurs when a class declares a method with the same signature as that of an inherited method
- You can still invoke the superclass' method with the syntax super.name(parameters).
- Restrictions
  - Overriding method must have the same return type as the method it overrides
  - The overriding method cannot be more private than the method it overrides (public > protected > package > private).
  - The overriding method may not throw any exception types in addition to those thrown by the method it overrides
- A class can declare a variable with the same name as an inherited variable, thus "hiding" or shadowing

Inheritance

- Enables reusability of a defined type
- A class can extend another class
  - Inherits all the data members and methods
  - The new class(child) can redefine the parent methods and/or add its own methods
- Inheritance implements a “is a” relationship
- Two types of inheritance
  - Single (supported by Java)
  - Multiple (Kind of supported)

Inheritance – Terminology



subclass *or* derived class — extends — superclass *or* base class

subinterface — extends — superinterface

class — implements — interface

Constructors make objects

- Every class has a constructor to make its objects
- Use the keyword new to call a constructor

    secretary = new Employee ( );

- You can write your own constructors; but if you don’t,
- Java provides a default constructor with no arguments
    - It sets all the fields of the new object to zero
    - If this is good enough, you don’t need to write your own
- The syntax for writing constructors is almost like that for writing methods

Syntax for constructors

- Instead of a return type and a name, just use the class name
- You can supply arguments

```
Employee (String theName, double theSalary) {
   name = theName;
   salary = theSalary;
}
```

Use the same name for a parameter as for a field

- A parameter overrides a field with the same name
- But you can use this.name to refer to the field

```
Person (String name, int age) {
   this.name = name;
   this.age = age;
}
```

- This is a very common convention

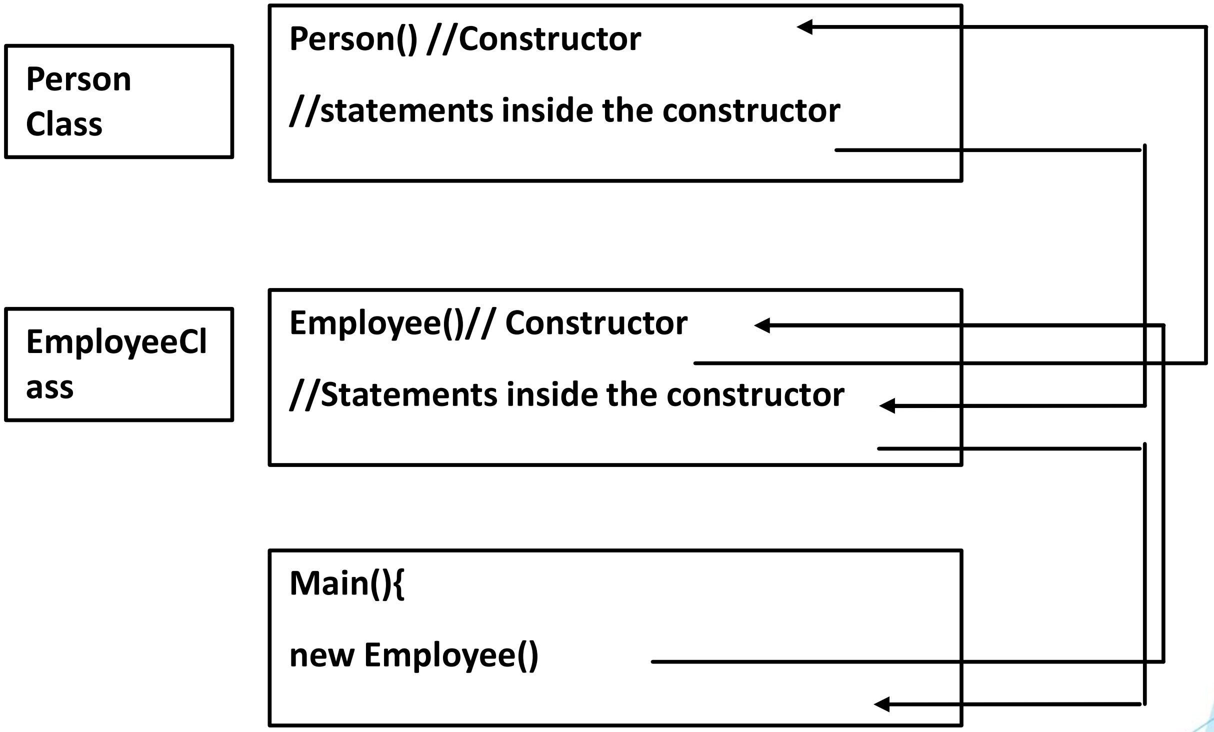
Constructor chaining

- If an Employee is a Person, and a Person is an Object, then when you say new Employee ()
  - The Employee constructor calls the Person constructor
  - The Person constructor calls the Object constructor
  - The Object constructor creates a new Object
  - The Person constructor adds its own stuff to the Object
  - The Employee constructor adds its own stuff to the Person

Constructor Chaining

- A subclass constructor invokes the constructor of the super class implicitly
  - The default constructor of the super class
- A subclass constructor can invoke the constructor of the super explicitly by using the “super” keyword
  - Used when passing parameters to the constructor of the super class

Constructor calling



Person Class

Person() //Constructor

//statements inside the constructor

EmployeeClass

Employee()// Constructor

//Statements inside the constructor

Main(){

new Employee()

The case of the vanishing constructor

- If you don't write a constructor for a class, Java provides one (the *default constructor*)
- The one Java provides has no arguments
- If you write *any* constructor for a class, Java does *not* provide a default constructor
- Adding a perfectly good constructor can break a constructor chain
- You may need to fix the chain

Example: Broken constructor chain

```
class Person {
   String name;
   Person (String name) { this.name = name; }
}
class Employee extends Person {
   double salary;
   Employee ( ) {
      // here Java tries to call new Person() but cannot
find it;
      salary = 12.50;
   }
 }
```

Fixing

- Special syntax: super(...) calls the super class constructor
- When one constructor calls another, that call *must be first*

```
class Employee {
 double salary;
    Employee (String name) {
        super(name);   // must be the first statement

        salary = 12.50;
    }
}
```

- Now you can only create Employees with names
- This is fair, because you can only create Persons with names

Abstract Methods

- Methods that do not have implementation (body) are abstract
- To create an abstract method use the keyword abstract and no definition
  - Not void someMethod(){ }
- For example
  - public abstract void someMethod();

Abstract Classes

- An abstract class is one that contains one or more abstract methods
  - It must itself be declared with the abstract keyword
- A class may be declared abstract even if it does not contain any abstract methods
- A non-abstract class is sometimes called a concrete class.
- An abstract class cannot be instantiated
- Instead, you can create subclasses that implements the abstract methods of the base class

Interface

- A contract agreed by the class
- The class agrees to implement the methods declared by the interface
- All methods in interface are abstract.
- The only fields that can appear in an interface must be declared both static and final
- All members are public by default
- Methods & Interface is implicitly abstract
- A concrete class must implement the interface
  - Implement all the abstract methods of the Interface

Why to use Interfaces?

- To reveal the programming interface of an object without revealing its implementation
  - This is the concept of encapsulation
  - The implementation can change without affecting the caller of the interface
- To have unrelated classes implement similar methods
- A class can implement multiple interfaces while it can extend only one class

Abstract class vs Interface

Abstract Class	Interface
Can extend only single base class	Can implement any number of interfaces
Can have concrete methods	All methods are abstract
Little faster	Involves search before calling
More suited for code reuse	Suitable for type declaration
Adding new methods does not cost much. You could have a default implementation.	Adding new method involves defining that in all the implementing classes

Packages

- A package is a grouping of related types
  - types like classes, interfaces, enumerations, and annotation types
- Bundling the classes and the interface in a package to
  - Easily determine that these types are related
  - Know where to find types that can provide graphics-related functions
  - Avoid naming conflicts
  - Allow types within the package to have unrestricted access to one another

Creating a Package

- Choose a name for the package
  - Package names are written in all lower case
  - Companies use their reversed Internet domain name
    - com.example.mypackage for a package created at example.com
- put a package statement with that name at the top of every source file
- The package statement must be the first line in the source file
- There can be only one package statement in each source file
- Syntax: package <package name>;

Using Package Members

- Refer to the member by its fully qualified name
  - Ex: <package Name>.<Class Name>
- Import the package member
  - import <package Name>.<Class Name>;
- Import the member's entire package
  - import <package Name>.*;

Summary

- OO programming involves three fundamental concepts: ADTs, inheritance, dynamic binding
- A Java program is a collection of classes