

# SOFTWARE ENGINEERING I

Java objects, classes, and interfaces

[andrea.corradini@mci.edu](mailto:andrea.corradini@mci.edu)



## Tentative topics

- Webinar 1: NetBeans IDE and Maven, Java language I: Java objects, classes, and interfaces
- Webinar 2: Java language II
- Webinar 3: design patterns I
- Webinar 4: design patterns II
- Webinar 5: design patterns III
- Webinar 6: OOP principles

# Classes and objects

- A class is a sort of blueprint that defines each class's object properties and behaviors
- An object is an instance of a class, hence more objects can be created out of a class
- The idea behind OOP is to model a problem in terms of objects as abstractions/conceptualizations of real world entities that cooperate with each other

# Modeling a bank account

- A class *BankAccount* describes an account at any bank: it has a balance and an ID (data fields); the balance can be queried, and money can be withdrawn from or added on it (methods)
- My account at Raiffeisen, your account at PostBank are two possible instances (i.e. objects) of the class *BankAccount*

# BankAccount Class (1<sup>st</sup> Version)

```

public class BankAccount {           // by convention class names are capitalized
    int number;                      // data field/instance variable
    int balance;                     // data field/instance variable

    public BankAccount(int assignedNumber) {    // constructor
        number = assignedNumber;
        balance = 0;
    }

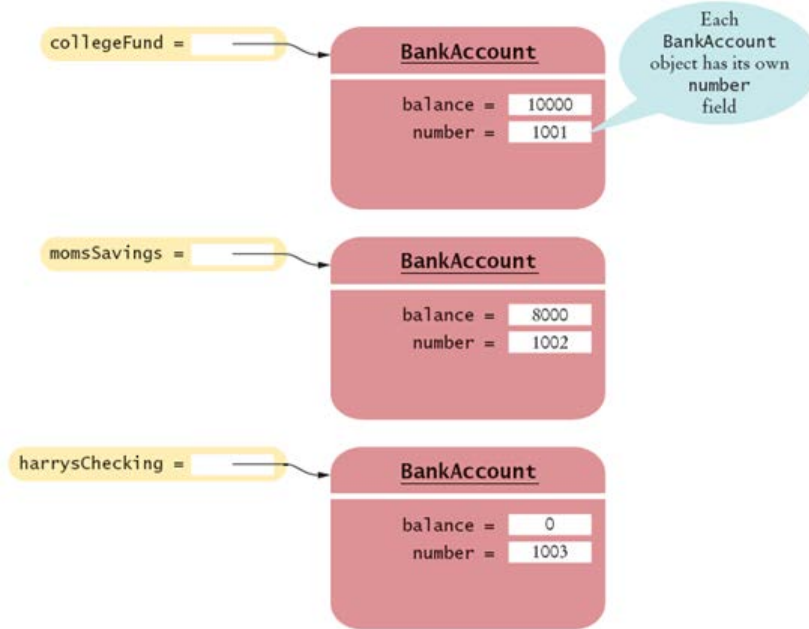
    public void deposit(int amount) {           // instance method
        balance += amount;
    }

    public int getBalance() {                   // instance method
        return(balance);
    }
}

```

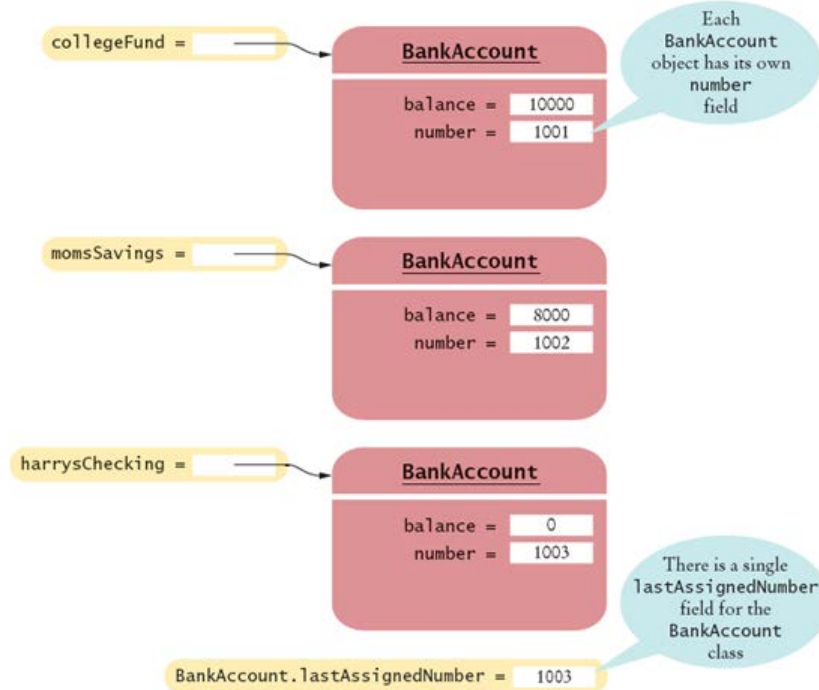
# Instance data fields

- Data members common to all objects of a class are referred to as *instance data members* or *instance data fields* or *non-static fields* or *instance variables*
- Instance variables belong to a specific instance
- Problem: where can one store the total number of opened accounts?
- Solution 1 (bad) : use an instance data fields that store a copy of the value in every object



# Static data fields

- Solution 2: use a field that is shared by all instances of the class
- Data members peculiar to the class are called *class data members* or *static fields* and are declared with the *static* keyword preceding the type name



# BankAccount Class (2<sup>nd</sup> Version)

```

public class BankAccount {                                // by convention class names are capitalized
    private static int lastAssignedNumber = 0;           // static data member: shared by all
                                                         // BankAccount objects

    private int number;    // data field/instance variable
    private int balance;   // data field/instance variable

    public BankAccount() {                             // constructor
        ++ BankAccount.lastAssignedNumber;
        number = BankAccount.lastAssignedNumber;
        balance = 0;
    }

    public void deposit(int amount) { ...// same instance method as before
    public int getBalance() { ...      // same instance method as before
}

```



# Static methods vs instance methods

- Similarly to data members, methods can be of two types
  - *class* or *static methods* may be called even when no object of a class exists; they are not tied to a specific object, thus they cannot refer to instance data members; they are declared with the *static* keyword and invoked via the class name
  - *instance methods* are methods specific of a given object, hence are invoked by an instance of the class
- That is why *main()* must be declared as a *static* method: it is the entry point of any application and as such has to be executed even if there is no object around

# Accessing data members and methods

- The dot operator on a object reference variable *objRefVar* for class *ClassName* is used to access instance data members as well as to call a non-static method of that object:

*objRefVar.dataMember; // access an instance data member*  
*objRefVar.instanceMethod(args); // run an instance method*
- The dot operator on the class name *ClassName* is used to access static data members as well as to call a static method of that class:

*ClassName.staticDataMember; // access a static data member*  
*ClassName.classMethod(args); // run a static method*
- A set of access/visibility modifiers sets the rights for accessing methods and data members

# Constructors

- A constructor has the same name as the class it is defined in
- A constructor contains the code that runs when an object is instantiated
- A constructor looks like a method but it is not a method:
  - it cannot be called on an object like methods are
  - may be invoked only using the *new* keyword followed by the class name
  - it does not have any return type
  - may not be declared *static*: no sense to disassociate a constructor from its object!
  - may not be declared *final* or *abstract*

# Default constructor

- A constructor creates an instance of a class
- Every class has a constructor, even if not explicitly provided, because Java provides a default no-arg constructor:

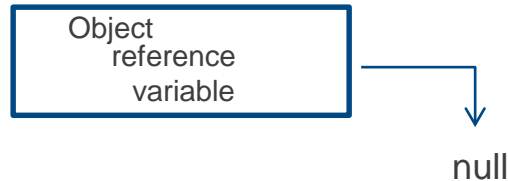
```
public BankAccount() {  
}
```

- If the coder provides at least one constructor (overloading is allowed), no default constructor is added by the compiler
- Constructors may have arguments:

```
public BankAccount(int numberAccount) {  
    number = numberAccount;  
}
```

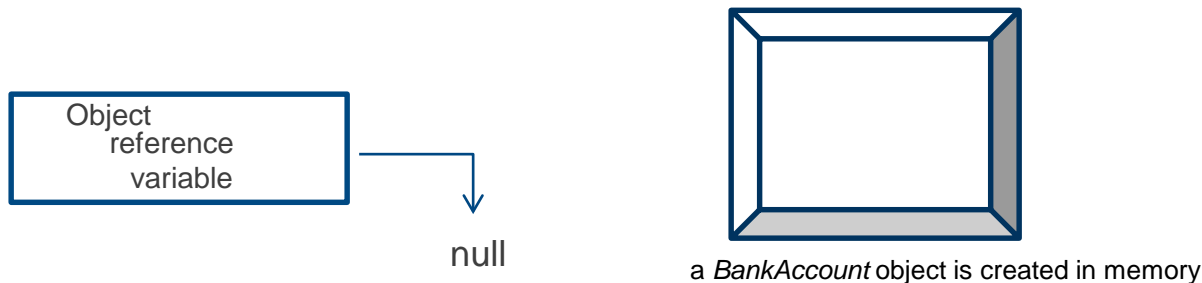
# Constructing object

- Construct an object with the *new* statement:  
*BankAccount myAccount = new BankAccount();*
- Three steps:
  - declare an object reference variable (by default initialized to *null*)



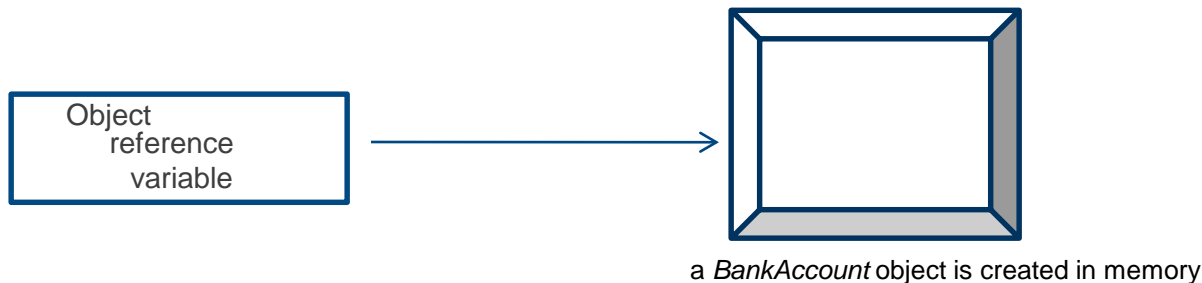
# Constructing object

- Construct an object with the *new* statement:  
*BankAccount myAccount = new BankAccount();*
- Three steps:
  - declare an object reference variable (by default initialized to *null*)
  - call the *new* statement with a method-like *BankAccount()* thingy to create an object in memory



# Constructing object

- Construct an object with the *new* statement:  
*BankAccount myAccount = new BankAccount();*
- Three steps:
  - declare an object reference variable (by default initialized to *null*)
  - call the *new* statement with a method-like *BankAccount()* thingy to create an object in memory
  - link the object reference variable to the newly created object



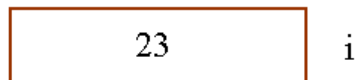
## References to objects

- Objects and arrays are reference types i.e. handles that point to the memory location of the object
- By defining variable *myAccount* of type *BankAccount* one is not creating an object of type *BankAccount* but rather is defining a reference to such an object
- More handles can reference to the same object
- If an object is not pointed to by any reference, the object is not accessible anymore

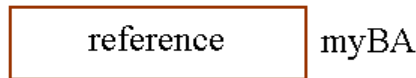


# Primitive types vs object types

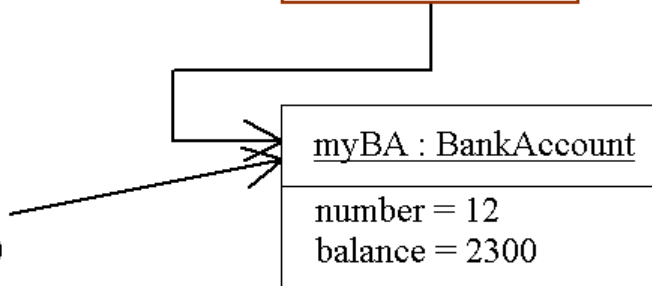
Primitive type: `int i = 23`



Object type: `BankAccount myBA`



Created using  
`new BankAccount()`



- Primitive types are stored as values

*`int i = 23, k;`*

*`k = i; // there are two copies of  
// value 23 in memory!`*

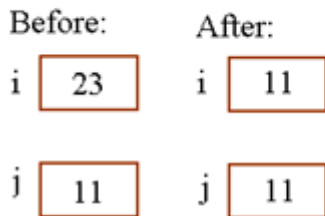
- Reference type variables are stored as references

*`BankAccount b, myBA = new BankAccount();`*

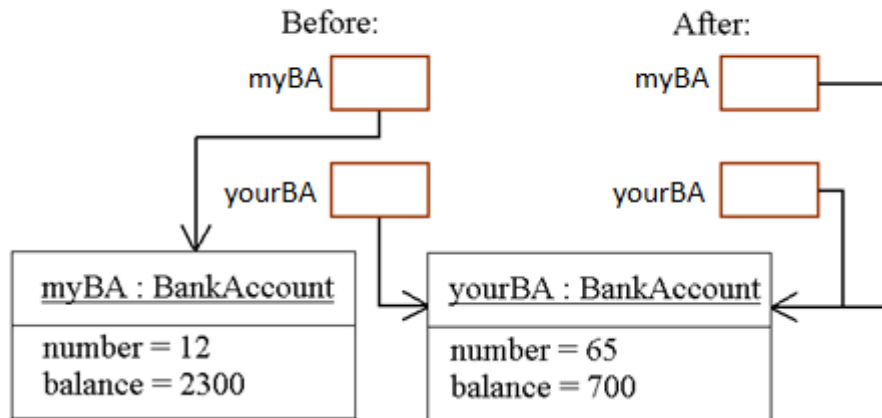
*`BankAccount b = myBA; // there is only one  
// object in memory`*

# Assigning primitive types & object types

Primitive type assignment:  $i = j$



Object type assignment:  $b1 = b2$



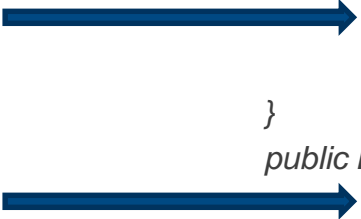
# The variable *this*

- Within an instance method or a constructor, *this* is used to refer to the current object
  - every instance method and constructor has a variable called *this*
  - hence *this* can exist only if an object actually exists
  - the Java compiler uses it implicitly to refer to an instance variable as well as an instance method from within an instance method or a constructor
- Used to invoke other constructors of the object from inside another constructor
- One should not clutter up a program with lots of *this* unless necessary for disambiguating

# Calling a constructor from a constructor

- A constructor can be called from another constructor using the *this* keyword as the method name that links the two constructors

```
public class BankAccount() {
    public BankAccount() {
        System.out.println("Creating a new bank account!");
    }
    public BankAccount(int numberAccount) {
        this();
        number = numberAccount;
    }
    public BankAccount(int numberAccount, String nameOwner) {
        this(numberAccount);
        name = nameOwner;
    }
}
```



# Getters and setters

- Setters/getters are mutators methods used to set/get instance variables; getters/setters returns/sets the value of instance variables
- A most common reason for using the *this* keyword is because a field is shadowed by a method or parameter

```
public class BankAccount {
    private int number;
    public int getNumber() {    // Getter; by convention it is called getVariableName
        return number;        // With boolean values, getters are called isVariableName
    }
    public void setNumber(int number) {    // Setter; by convention it is called setVariableName
        this.number = number;    // used to disambiguate the variable number
    }
}
```

# Method overloading

- Several methods with the same name are allowed as long as each method has a unique set of parameters
- Defining multiple methods with the same name in a class is called *method overloading*: resolution occurs at compile-time
- Method name along with types and sequence of parameters constitute the signature of a method
- Signature of two overloaded methods must be distinct for the compiler to determine uniquely the method called at any point
- The return type has no effect on the signature: one cannot differentiate between methods just by return type for this is not necessarily apparent at method call

# Method overloading: example

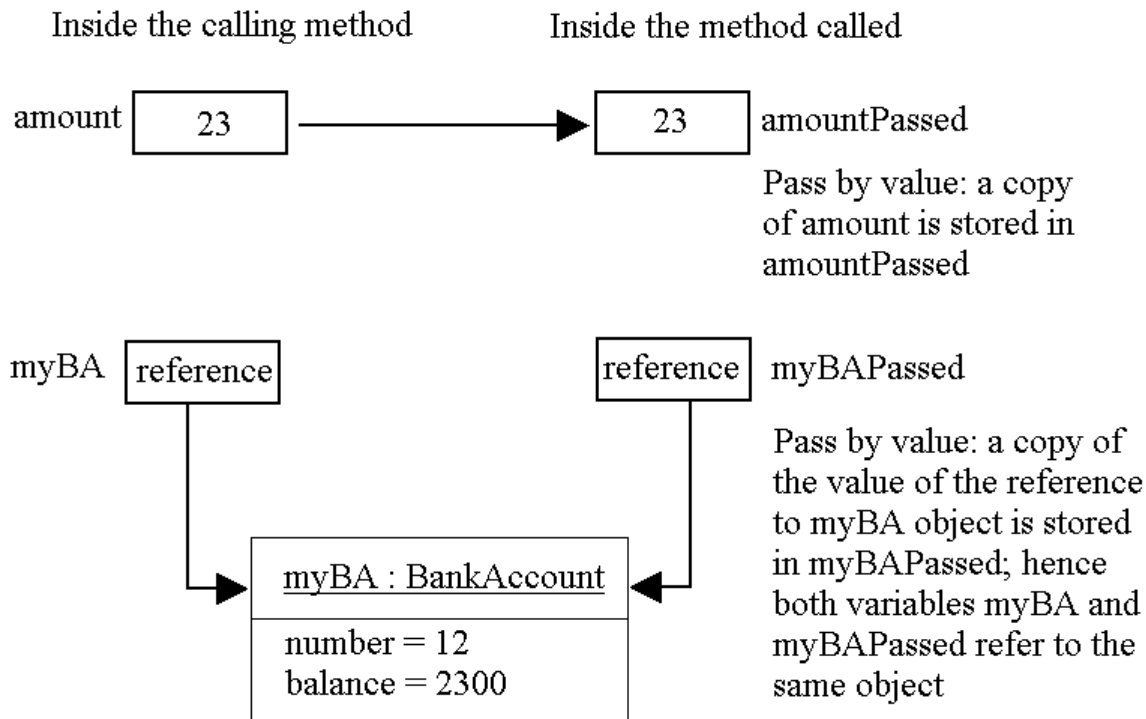
```
public class BankAccount {  
    private String msg;  
  
    public void printNumber() {        // No arguments  
        System.out.print(msg);  
    }  
  
    public void printNumber(String postfix) {    // One arguments  
        System.out.println(msg + postfix);  
    }  
  
    public void printNumber(String prefix, String postfix) {    // Two arguments  
        System.out.print(prefix + msg + postfix);  
    }  
}
```

# Passing parameters to methods (I)

- A primitive data type variable is passed as an argument to a method by value
  - a copy of the variable is created and passed on to the method
  - the called method can not modify the value of the original variable passed on to the method
- An reference to an object is passed as an argument to a method by value
  - a copy of the reference contained in the variable (i.e. the value of the reference to the object!) is transferred to the method, not a copy of the object
  - the copy of the reference passed refers to the original object, thus the called method can modify the original object



# Passing parameters to methods (II)

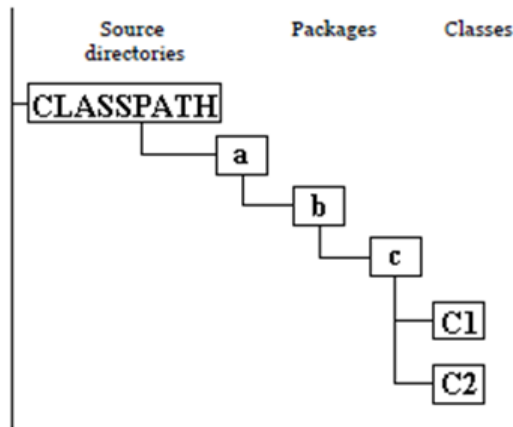


# Packages

- Packages make
  - types easier to find and use
  - it possible to avoid naming conflicts and collisions (namespace management)
  - it possible to control access since packages provide a level of security through access modifiers
  - possible grouping and bundling of related types (class, interface, enumerations, annotation)
- Example of API packages are *java.lang*, *java.util*, etc.
- To create a package, a package statement with that name must be put at the top of every source file to include in the package  
*package at.mci.sweng1.week1.andrea;*

# Packages: access

- By convention package names are in lower case and use the (unique) reverse Internet domain like *at.mci.andrea* or *a.b.c*
- Package names reflect the path to the folder where their classes are stored i.e. *a.b.c.C1* and *a.b.c.C2* are two classes in package *a.b.c*
- Class *a.b.c.C1* can be referred
  - using their its fully qualified name  
*a.c.c.C1 myC = new a.b.c.C1();*
  - by importing the entire package  
*import a.b.c.\*;*  
*C1 myC = new C1();*
  - by importing the class only  
*import a.b.c.C1;*  
*C1 myC = new C1();*



# The *import* statement

- Java comes with a collection of classes organized in packages, called Java Class Library or Java Application Programming Interface (API)
- The *import* statements is needed to use a class from a particular package

```
import java.util.*;  
import java.util.ArrayList;
```

- It does not make the class bigger i.e. does not compile imported classes and packages into the code

## *import* and the Java API

- It allows to reference the classes using only the class name
- Without *import* statement one has to specify the fully qualified name of each class from a package to refer to it
- Import or type in, because these are equivalent:

```
import java.util.ArrayList;  
ArrayList myArray = new ArrayList(0);
```

or

```
java.util.ArrayList myArray = new java.util.ArrayList(0);
```

- The documentation in HTML for the Java API can be found at:  
<https://docs.oracle.com/en/java/javase/21/docs/api/index.html>
- One can create the JavaDoc of the own classes

# The package *java.lang*

- Important classes and interfaces that are imported automatically in all source code as if the compiler inserted into any source *import java.lang.\*;*
- Specifically
  - *java.lang.String*
  - *Java.lang.Math*
  - *java.lang.Object*
  - *java.lang.Integer* and other wrappers
  - *java.lang.System* contains a hodgepodge of methods, most of which relate to advanced JVM functionalities

# The class *java.lang.Object*

- *Object* is the root of the Java class hierarchy i.e. it is the superclass of any other class
- All objects implement the methods of this class:
  - *wait()*, *notify()*, *notifyAll()*: all for thread control
  - *equals()*
  - *toString()*
  - *hashCode()*
  - *getClass()*
  - *finalize()*
  - *clone()*

# Access/Visibility modifiers

- Accessibility to classes, data members and methods is controlled by *access attributes*
- There are four access level to a class, data members, and methods, but only three access modifiers
  - *public*: the class, data, or method is visible to any class in any package
  - no explicit access attribute specified (called default or package-private): accessible from any class in the same package
  - *protected*: like the default modifier, accessible from any class in the same package and from any subclass of the declaring class
  - *private*: accessible only by the declaring class



# Access level

Modifier	Class ( <i>Alpha</i> )	Package ( <i>Beta</i> )	Subclass of a class, also declared outside the package ( <i>AlphaSub</i> )	World ( <i>Gamma</i> )
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

# Abstract methods and abstract classes

- An abstract method is a method with an abstract modifier and no implementation
- An abstract class is a like a regular class that may contain abstract methods and with an *abstract* modifier
- One may not instantiate an abstract class with *new* yet may define its constructors that are invoked by its subclasses' constructors

```
public abstract class MyAbstractClass {  
    // data members  
    public MyAbstractClass() {  
        // constructor body  
    }  
    // instance methods (i.e. they have a body)  
    abstract protected void pettyPrint(); // abstract method has no method body  
}
```

# Why abstract classes?

- To model a real-world problem by creating several classes that have some common functionalities
- A *Chart* class that has
  - common functionalities like e.g. *usePalette()*, *useBGColor()*
  - unique functionalities like *displayChart()* depend on chart type
- Extending *Chart* e.g. into subclasses *PieChart* or *BarChart* each representing a type of chart and sharing common functionalities but each implementing its own specific *displayChart()* is a good design choice

# Java Interface

- It is a class-like constructs that may not be instantiated with *new*

```
public interface MyInterface {
    int aConst = 23;           // it is public, static and final by default
    void anAbstractMethod();   // it is public and abstract by default
    static int aStaticMethod() { // it is public by default
        // must have a method implementation
    }
    default void aDefaultMethod() {
        // must have a method implementation
    }
    class ANestedType {
        // nested class body
    }
}
```

# Why interfaces?

- An interface is a special type of inheritance
- Specify common behaviors for objects that have a weak is-a relationship with the interface e.g. by being cloneable, comparable, etc.
- Define a standard set of operations which must be implemented by the classes using it
  - objects of different classes implementing an interface share the common set of operations (usually with different implementation)

# Extending or implementing interfaces

- A Java class may implement any number of interfaces

```
public class MyClass implements Interface1, Interface2 {
    public void implementMe() { // here method implementation }
    // all methods declared in Interface1 and Interface2 must be implemented here!
}
```

- An interface may not implement an interface but it may extend any number of (super) interfaces thus simulating multiple inheritance:

```
public interface MyInterface extends Interface1, Interface2 {
    public void implementMe();
    // no need to implement the methods of Interface1 and Interface2 here!
}
```

# Abstract classes vs. interfaces

- Abstract Class
  - May not be instantiated by *new*
  - May define constructors
  - Constructors may not be *abstract*, *final*, or *static*
  - No restrictions on data fields and non abstract methods
  - Abstract methods may not be *private*, *final*, or *static* otherwise cannot be inherited
  - Can extend one class only
  - May implement interfaces
- Interface
  - May not be instantiated by *new*
  - May not define constructors
  - Data fields are *public static final* i.e. constants
  - Methods can be *abstract* or *default* or *static*
  - Methods may not be *private*, *protected*, or *final*
  - Can extend multiple interfaces
  - Multiple interfaces may be implemented by a class

# SOFTWARE ENGINEERING I



Java inheritance, polymorphism, and exception handling

`andrea.corradini@mci.edu`

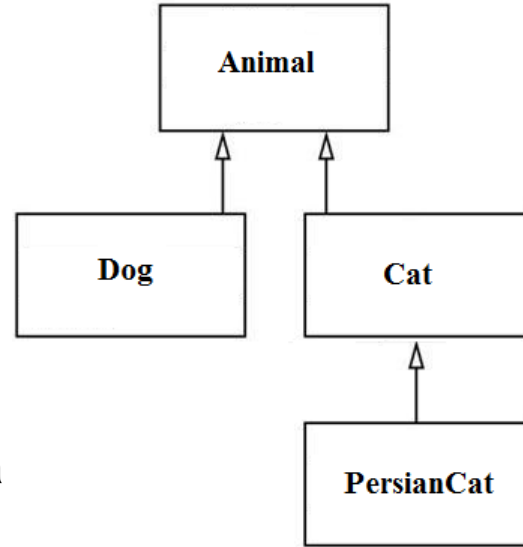


## Tentative topics

- Webinar 1: NetBeans IDE and Maven, Java language I: Java objects, classes, and interfaces
- Webinar 2: Java language II: Java inheritance, polymorphism, and exception handling
- Webinar 3: design patterns I
- Webinar 4: design patterns II
- Webinar 5: design patterns III
- Webinar 6: OOP principles

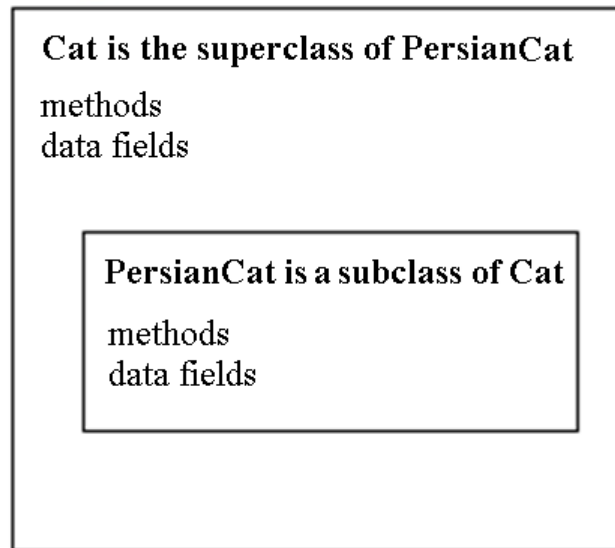
# Class inheritance

- How can one design classes that share common features and avoid redundancy?
- Inheritance allows to create a new class based on a class that is already defined
  - organizes classes so that common properties and behaviors can be defined only once for all classes involved
  - avoids code duplication since more specialized classes can be derived from a general class



# is-a relationships

- Class inheritance defines a binary, strong, unidirectional is-a relationship between two classes expressed by
  - “a *PersianCat* is a *Cat*” with few additions
- A weak *is-a* relation is expressed in Java with an interface
  - e.g. to find the best *Cat* one has to be able to compare *Cat* objects (done by implementing the *Comparable* interface)



# Inheritance in Java: *extends*

```
public class Cat {
    // data members of the Cat class
    protected String name;

    // constructors of the Cat class
    public Cat(String n) {
        name = n;
    }

    // other instance methods
}
```

```
public class PersianCat extends Cat {
    // data members of the PersianCat class
    private String breed;

    // methods of the PersianCat class
    public PersianCat(String n, String b) {
        ... }

    public void persianSpecializedTask() {
        ...
    }
}
```

- A class may extend only one other class!

# Derived class constructors and *super*

- A derived class constructor must initialize the base class
- Base class constructors are not inherited in derived classes
- Base class constructors are always invoked in a derived class either explicitly or implicitly
  - when not explicitly called, the no-arg base class constructor is invoked placing *super()* as first statement in the derived class constructor
  - when explicitly inserted, the call to the base class constructor via *super()* must be the first statement in the derived class constructor
- The dot operator applied to *super* is used to access data and methods of the super class from inside the child class

# Can a child exist before its parent?

- When a new object is created, all the constructors in the object's inheritance tree are run to build the superclass parts of the object
- The superclass parts of an object must be completely built before the subclass parts can be constructed and accessed

**NOPE!**

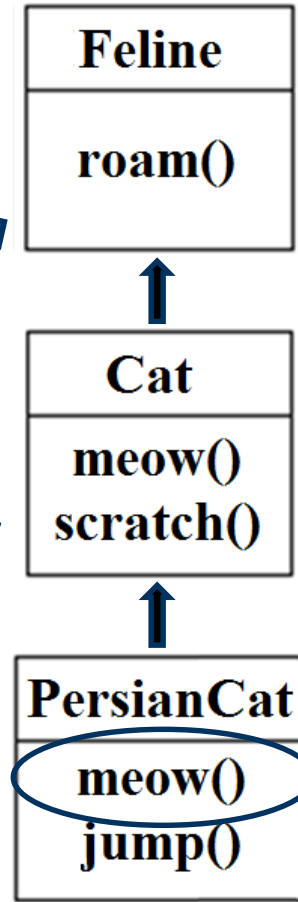
# Overriding methods

- A method of the base class can be overridden in a derived class if:
  - arguments are the same and return type compatible (covariant data type) i.e. they have the same signature
  - it is accessible from outside the class hence it may not be a *private* method
  - the access modifier of the overriding method may be different, but not more restrictive
- The overridden methods in a derived class are called, not the method inherited from the base class
  - the base class methods are still there and still can be called in the derived class via *super* keyword

# Calling overridden methods

- A *PersianCat* is a *Cat*, and a *Cat* is a *Feline*
- Not all *Feline* objects are *Cat*, and not all *Cat* objects are *PersianCat*
- Running this code produces:  

```
PersianCat mP = new PersianCat();
mP.roam();
mP.meow();
mP.scratch();
```





## Using *Object* as parameter type

- *Object* is the implicit superclass of each class
- It thus holds that a variable of type *Object* can store a reference to an object of any class type like in:  
`Object obj = new AnyClass(); // implicit casting`
- This is useful to write methods that need to handle objects of unknown type such as  
`public void someMethod(Object o)`
- Inside such a method though, only methods of *Object* can be called on the reference *o* unless *o* is explicitly casted

# The modifier *final* for methods

```
public class MyApplication {
    ...
    final public String about() {
        return("Copyright by Andrea");
    }
}
```

- Prevents *about()* to be overridden by a subclass of *MyApplication*

# The modifier *final* for classes

```
public final class MyApplication {
    ...
    public String about() {
        return("Copyright by Andrea");
    }
}
```

- Prevents the class itself to be subclassed e.g. *MyApplication* cannot be the base class of any class because it is declared *final*

# Inheritance and encapsulation

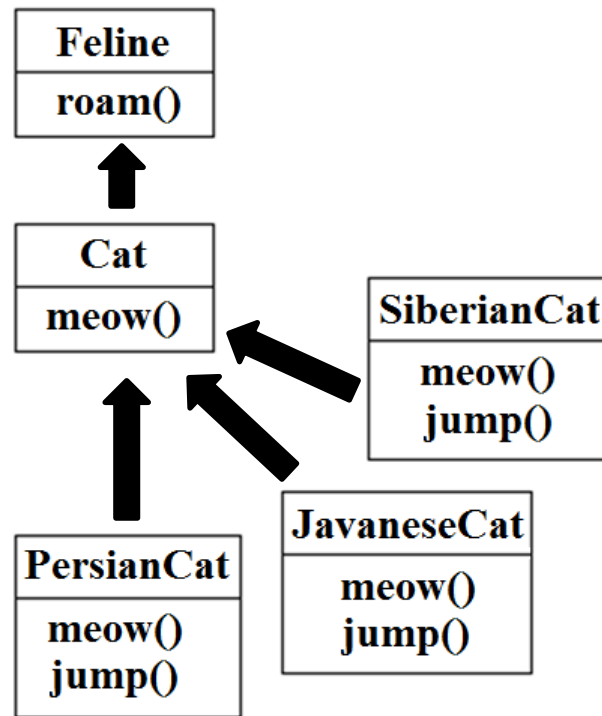
- Inheritance weakens encapsulation because changes in the superclass ripple through the class hierarchy
- Inheritance cannot handle changes well since it sets a strict “is-a” relationships between the base class and the derived classes
- Good OOP design separates the parts of the code that change the most from the rest of the application and makes them as independent as possible

# Inheritance vs composition

- Do not use inheritance whenever you could
  - *“if you only have hammer, everything looks like a nail”*
- When planning for change:
  - “has-a” relationships should be preferred over “is-a” relationships
  - volatile code should be put in the objects of the application, rather than inheriting that code
- By extracting the volatile part of the code and encapsulate it into other objects:
  - the code can be customized by creating composites of objects
  - the classes can be more easily updated as change happens

# Polymorphic behavior

- With polymorphism, one can refer to an object of a subclass with a reference to the base class  
*Cat myCat = new PersianCat();*  
*myCat.meow(); // calls the "right" method*
- Polymorphism applies to methods declared in the base class that are overridden in derived classes
  - meow()* is defined in *Cat* and *PersianCat*
- Which implementation is called is determined dynamically by the JVM at runtime (dynamic binding)



# Static type and dynamic type

- The static type of a variable is the type that appears in its declaration
- With declaration  
*Cat myCat;*  
*Cat* is the static type of *myCat*, it is determined at compile time
- The declared type defines which method to match at compile time based on the signature (matching)
- The type of object that a variable references at during runtime is its dynamic type
- A variable of reference type is a polymorphic one
  - the JVM binds the implementation of the method at runtime (binding)
  - the dynamic type, not the static, determines which method definition is called

# Overriding static methods

- Like instance methods, static methods can be inherited
- If a static method defined in the base class is redefined in a subclass, the method defined in the base class is hidden
- Static methods can therefore not be overridden, they can only be re-defined
- Since polymorphism only applies to overrides, invoking that method on a super type reference does not produce any polymorphic behavior



# Be the compiler and the JVM!

```
public class Cat {
    public void one() {
        this.two();
    }
    public void two() {
        System.out.println("Cat:two()");
    }
    public void three() {
        System.out.println("Cat:three()");
    }
}

public class PersianCat extends Cat {
    void one() {
        super.one();
    }
    void two() {
        System.out.println("PersianCat:two()");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Cat c = new Cat();
        c.one();
        PersianCat p = new PersianCat();
        p.one();
        p.three();
        c = p;
        c.one();
    }
}
```

- Output: Cat:two() - PersianCat:two() - Cat:three() - PersianCat:two()

# Polymorphic arguments

- If class *Feline* contains the method:

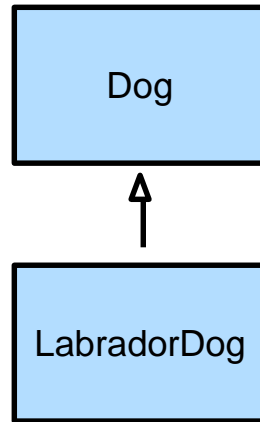
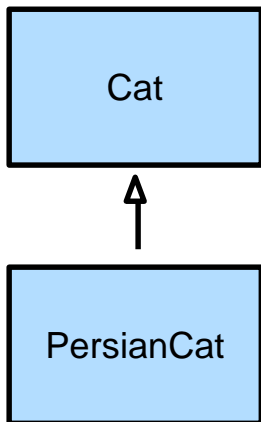
```
public static void foo(Feline feline) {  
    feline.meow();  
}
```

- A call to that method can exploit polymorphism:

```
Feline.foo(new Cat());           // runs Cat's meow()  
Feline.foo(new SiberianCat());  // runs SiberianCat's meow()
```

- Through polymorphism, if one adds a subclass the code does not have to be modified

# Method dispatching: ambiguity



- The following methods are declared in class *Zoo*:  
*play(Cat c, LabradorDog ld)*  
*play(PersianCat pc, Dog d)*

- These calls are either ambiguous or wrong in their arguments:  
*play(new PersianCat(), new LabradorDog());*  
*play(new Cat(), new Dog());*

# Typecasting

- Cast an object to another class type only if current object type and the new one are in the same class hierarchy
- Cast up a reference to an object of a class upwards through its direct and indirect superclasses:  
*Cat c = new PersianCat("Tom"); // explicit cast not necessary!*
- Cast down a reference to an object of a class downward through its direct and indirect subclasses:  
*PersianCat pc = new Cat("Tom"); // compilation error*  
*Persian pc = (PersianCat) new Cat("Tom"); // explicit casting necessary*  
*// yet run-time error*
- Ensure that an object is an instance of another object before attempting a cast operation

# The operator *instanceof*

- Given a variable *c* of type *Cat*, to cast it to type *PersianCat*:

```
Cat c = new Cat();
```

```
PersianCat pc;
```

```
if (c instanceof PersianCat) { // is cast possible?
```

```
    pc = (PersianCat) c; // now it is safe to cast!
```

```
    pc.someMethodOfPersianCat();
```

```
}
```

- instanceof* checks whether a reference is of a certain type or of any of its derived classes

# The method *getClass()*

- It is defined in class *Object* as *final public Class<?> getClass()* and returns an object of type *Class* that identifies the class of an object
- Example:

```
Cat c = new PersianCat();
Class catType = c.getClass();
if (catType == PersianCat.class) { // c is a PersianCat
    System.out.println(catType.getName());
} else {
    System.out.println("Should not get here!");
}
```

# Sub-classing and argument type

- This does not work:

```
public class Main {
    public static void foo(Object o) {
        o.meow(); // compile error
    }
    public static void main(String[] args) {
        Main.foo(new Cat());
        Main.foo(new PersianCat());
    }
}
```

- This does work:

```
public class Main {
    public static void foo(Object o) {
        if (o instanceof Cat) {
            Cat anyCat = (Cat) o;
            anyCat.meow();
        }
    }
    public static void main(String[] args) {
        Main.foo(new Cat());
        Main.foo(new PersianCat());
    }
}
```

# Interface and argument type

- We define interface *Meowable*:

```
public interface Meowable {
    void meow();
}
```

- We make *Cat* (automatically also *PersianCat*) implement it

```
public class Cat implements Meowable {
    // same class body as before
}
```

```
public class PersianCat extends Cat {
    // same class body as before
}
```

- We redefine *foo()*:

```
public class Main {
    public static void foo(Meowable o) {
        o.meow();
    }

    public static void main(String[] args) {
        Meowable myIntf = new Cat();
        myIntf.meow();
        Main.foo(new Cat());
        Main.foo(new PersianCat());
    }
}
```



# Polymorphism & inheritance prevention

- Subclassing might cause problems
- The reason to prevent inheritance is to ensure that the behavior of a class is not corrupted by any subclasses
  - polymorphism can easily be used to disrupt the behavior of a class
- If you design a class consider the implications of subclassing
- Declare the class *final* if you want to prevent inheritance

# Exceptions: rationale

- Exceptions signal unusual events in the code that deserve special attention
- Dealing with the many error conditions that might arise in the program concerning its normal operability, the code structure can soon become very complicated, thus exceptions
  - + separate the code that deals with errors from the code that executes when things are fine
  - + provide a way of enforcing a response to certain errors
  - + allow to handle runtime errors so that the program can continue to run or terminate gracefully
  - - require more time and resources since the handling mechanism requires instantiating new exception objects, rolling back the call stack, and propagating the errors to the calling methods

# Developer vs. programmer user (I)

- Class programmers:
  - Know the implementation and when error situations occur
  - Do not know what to do with the errors
  - Implement the class independently of any specific usage
- Class programmer's user:
  - Know that something might go wrong, but not where
  - Know how to handle the error
  - Use the class in a specific context

# Developer vs. programmer user (II)

```
public class Stack {
    List elements = new LinkedList();
    public void push(Object o) {
        elements.add(o);
    }
    public Object pop() throws EmptyEx {
        if (elements.size() == 0)
            throw new EmptyEx(this);
        return elements.removeLast();
    }
}

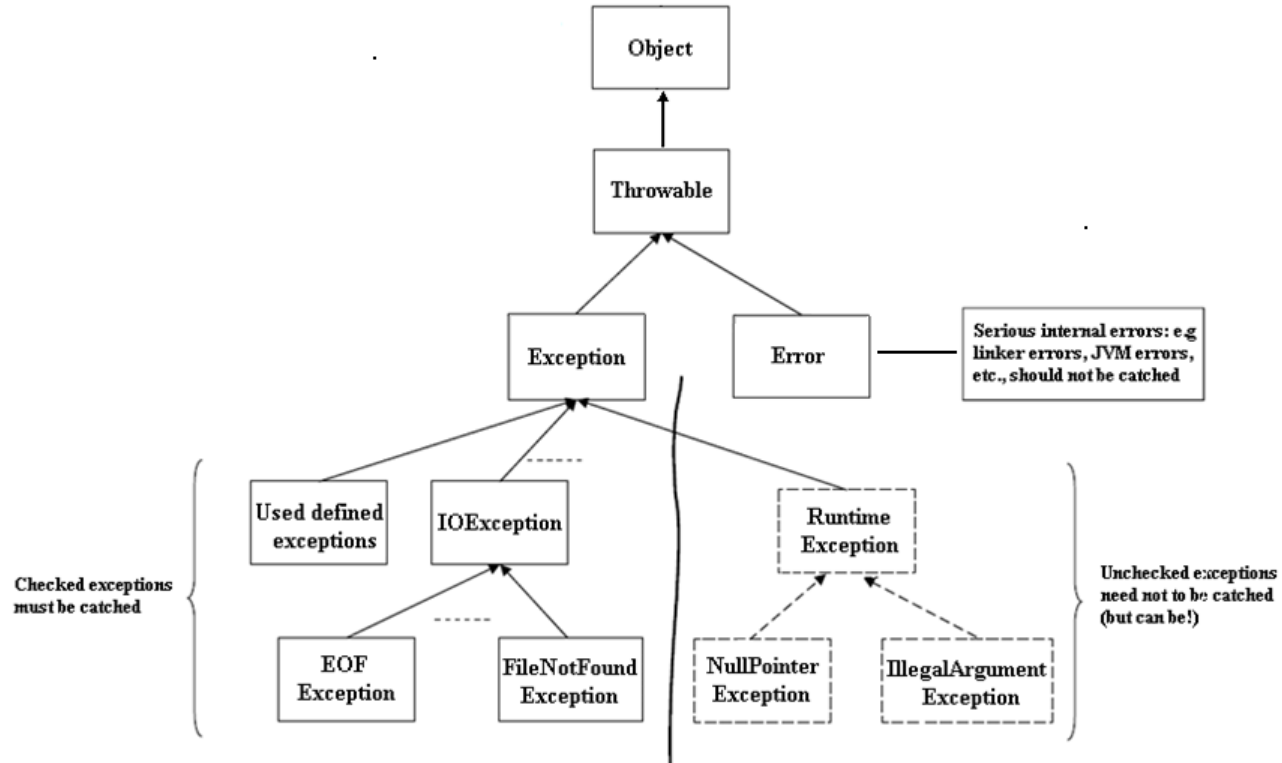
public class EmptyEx extends Exception {
    public EmptyEx(Stack st) { // do something
    }
}
```

```
Stack s = new Stack();
try {
    System.out.println( s.pop() );
}
catch (EmptyEx ex) {
    ...
}
```

# Java exceptions

- An exception in Java is an object that is created when an abnormal situation arises in the code and has fields that store information about the nature of the problem
- The exception is said to be thrown: the object that identifies the exceptional circumstance is tossed as an argument to a code specifically written to deal with it
- The code receiving the exception object as parameter is said to catch it
- If the calling code does not catch the exception, this is propagated up until it is handled in an exception handler in the code or is caught in the system application handler

# Exceptions: class hierarchy



# Checked & unchecked exceptions



- Unchecked exceptions:
  - *RuntimeException* and *Error* along with their subclasses
  - the compiler does not force the coder to check and deal with them: s/he can do so if s/he wants
  - reflects logical errors that cannot be recovered
  - it is not practical that correct code is polluted with error checks; all call to an instance method may potentially trigger a *NullPointerException*!
- Checked exceptions:
  - all the other exceptions
  - the compiler forces the coder to check and deal with them

# Declaring, raising and catching exceptions

- Every method must state the types of checked exceptions it might throw:

*public void myMethod() throws IOException, SomeOtherException*

- An exception can be raised using the construct *throw*:

*throw new Throwable("Throwing an exception");*

- The caller must capture the exception in a *try-catch* block:

```
try {
    objRef.myMethod();
    // here the call to the code containing the instruction above
} catch (Throwable t) {
    // handler for the exception: deal with the exception
}
```



# Dealing with checked exceptions

- Java forces to deal with checked exceptions
- If a method declares a checked exception, like e.g.  
`public void foo() throws IOException;`  
the exception must be
  - invoked in a *try-catch* block (case 1 below)
  - declared to throw the exception in the calling method (case 2 below)

```
public void bar() {
    try {
        foo();
    }
    catch (IOException ex) {
        ...
    }
}
```

(1)

```
public void bar()
    throws IOException {

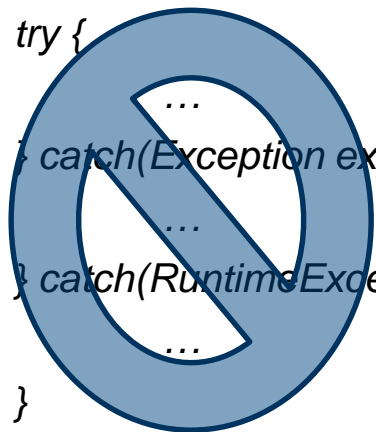
    foo();

}
```

(2)

# Order of catch blocks

- The order in which exceptions are specified in *catch* blocks is very important
- If a *catch* block for a superclass type is placed before a *catch* block for a subclass type, a compilation error occurs



```
try {
    ...
} catch(Exception ex) {
    ...
} catch(RuntimeException ex) {
    ...
}
```

```
try {
    ...
} catch(RuntimeException ex) {
    ...
} catch(Exception ex) {
    ...
}
```

# Catching any exceptions

- How can one catch any exception?
- In principle, one should catch a *Throwable* exception since all exceptions derives from class *java.lang.Throwable*
- Yet, exceptions in *java.lang.Error*, derived from *Throwable*, are used by the JVM and should not be caught in the code, hence to deal with application and runtime exceptions one usually catches *java.lang.Exception* exceptions

```
try {
    ...
} catch (Exception ex) {
    ...
}
```

## The clause *finally* I

- Anytime an exception is thrown, the execution of the *try* block code breaks off, regardless of the code that follows
- There is the possibility that the exception leaves things in an unsatisfactory state such as having open files, etc.
- The *finally* block provides the means to clean up at the end of executing a *try* block: it makes sure that some particular code is run before the method returns
- A *finally* block is always executed, regardless of whether or not an exception is thrown during the execution of the associated *try* block

## The clause *finally* II

- If no exception is thrown in the *try* block, whatever is in the *finally* is executed; the statement after the *try* block is executed
- If a value is returned within a *finally* block, this *return* overrides any *return* statement executed in the *try* block
- If exception thrown in *try* block is not caught in any *catch*, the other statements in *try* block are skipped, *finally* clause executed, and control propagated up to caller
- If exception is caught, the other statements in *try* block are skipped, *catch* block executed, and *finally* executed
  - if *catch* block throws an exception, control passed to caller
  - if not, next statement after *try* is executed

# Checked exceptions & inheritance



```
public class A {  
    public void do() throws Ex {  
        System.out.println("A:do()");  
    }  
}  
  
public class SubA1 extends A {  
    public void do() throws Ex, Ex1 {  
        System.out.println("SubA1:do()");  
    }  
}  
  
public class SubA2 extends A {  
    public void do() {  
        System.out.println("SubA2:do()");  
    }  
}
```

```
public void test(A a) {  
    try {  
        a.do();  
    } catch (Ex ex) {  
        ...  
    }  
}
```

- The exceptions *Ex* and *Ex1* are not in the same class hierarchy
- Does *test()* compile? And the other classes?
- Can the whole code run? What does it print out?

# Checked exceptions & inheritance



```
public class A {  
    public void do() throws Ex {  
        System.out.println("A:do()");  
    }  
}  
  
public class SubA1 extends A {  
    public void do() throws Ex, Ex1 { // illegal  
        System.out.println("SubA1:do()");  
    }  
}  
  
public class SubA2 extends A {  
    public void do() { // legal  
        System.out.println("SubA2:do()");  
    }  
}
```

```
public void test(A a) {  
    try {  
        a.do();  
    } catch (Ex ex) {  
        ...  
    }  
}
```

- ▶ The compiler accepts *test()*, because all checked exceptions thrown by *do()* are caught
- ▶ If method *do()* in *SubA1* would be allowed to throw an *Ex1* exception, this could not be caught in *test()*, hence one may not add new exceptions in overridden methods
- ▶ Yet, one may remove exceptions

# Checked exceptions & inheritance



```
class Ex1 extends Ex
class Ex2 extends Ex
class Ex3 extends Ex1
...
public void dolt() throws Ex1, Ex2 { // legal!
    ...
}

public void a() { // does it compile?
    try {
        dolt();
    } catch(Ex1) { ...
    } catch(Ex2) { ...
    } catch(Ex3) { ...
    }
}
```

```
public void a() { // does it compile?
    try {
        dolt();
    } catch(Ex1) { ...
    } catch(Ex3) { ...
    } catch(Ex2) { ...
    }
}

public void a() { // does it compile?
    try {
        dolt();
    } catch(Ex2) { ...
    } catch(Ex3) { ...
    } catch(Ex1) { ...
    }
}
```



# Checked exceptions & inheritance



```
class Ex1 extends Ex
class Ex2 extends Ex
class Ex3 extends Ex1
```

```
...
```

```
public void dolt() throws Ex1, Ex2 { // legal!
```

```
...
```

```
}
```

```
public void a() { // does not compile!
```

```
try {
```

```
    dolt();
```

```
    } catch(Ex1) { ...
```

```
    } catch(Ex2) { ...
```

```
→ } catch(Ex3) { ... // already caught!
```

```
}
```

```
}
```

```
public void a() { // does not compile!
```

```
try {
```

```
    dolt();
```

```
    } catch(Ex1) { ...
```

```
→ } catch(Ex3) { ... // already caught!
```

```
    } catch(Ex2) { ...
```

```
}
```

```
}
```

```
public void a() {
```

```
try {
```

```
    dolt();
```

```
    } catch(Ex2) { ...
```

```
    } catch(Ex3) { ...
```

```
    } catch(Ex1) { ...
```

```
}
```

```
}
```

# Re-throwing & chained exceptions

- Exceptions can be re-thrown by exception handlers to give a chance to the caller to process them

```
catch (Exc1 ex) {  
    // do something  
    throw ex;  
}
```

- A new exception can be thrown along with original one with (or without) a message info about this latter

```
catch (Exc1 ex) {  
    // do something  
    throw new Exception("Info", ex); // Exception(String, Throwable)  
}
```

# Custom exception classes

- Exception classes in the API should be used whenever possible but custom exception classes may be used if the predefined classes are not sufficient
- Custom exception classes are declared by extending *Exception* or any of its subclasses
- If the exception is bound to a specific class, the exception class can be made as an inner class of that class
  - if one does not need the *this* reference in instances of the exceptions, the exception class can be made static
- If the exception can occur in a range of different classes in a package, the exception class should be declared *public*

# SOFTWARE ENGINEERING I

## Design Patterns I

[andrea.corradini@mci.edu](mailto:andrea.corradini@mci.edu)

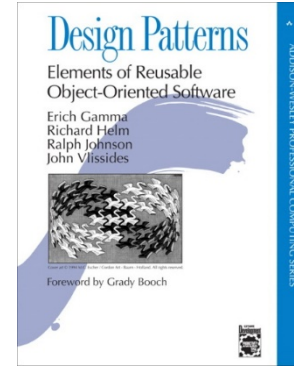


## Tentative topics

- Webinar 1: NetBeans IDE and Maven, Java language I: Java objects, classes, and interfaces
- Webinar 2: Java language II: Java inheritance, polymorphism, and exception handling
- Webinar 3: design patterns I: selected creational patterns
- Webinar 4: design patterns II
- Webinar 5: design patterns III
- Webinar 6: OOP principles

# What are design patterns?

- A general, repeatable solution to a common software problem
- *“...descriptions of communicating objects and classes customized to solve a general design problem in a particular context...”*
- *“...help you build on the collective experience of skilled software engineers...”*
- *“...capture existing, well-proven experience in software development and help to promote good design practice...”*
- *“... deal with a specific, recurring problem in the design or implementation of a software system...”*
- *“... can be used to construct software architectures with specific properties...”*



Erich Gamma et al. “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995

# What are design patterns NOT? (I)

- They are not algorithms or data structures
  - provide solutions to computational problems; patterns focus on non-functional issues like extendibility and maintainability
- They are not programming idioms
  - they are reoccurring low-level solutions to common programming problems in a specific language; design patterns are high-level and language independent
  - idioms are looked at while coding; patterns at design time
- They are not frameworks
  - design patterns are more general than frameworks and generative; frameworks are architectures that cannot generate solutions

# What are design patterns NOT? (II)

- They are not APIs or class libraries
  - they are solution written for a specific language / framework
- They are not principles or guidelines
  - they are more general than patterns because they can be applied to many different kind of problems; patterns are meant to solve a specific problem
  - strategies and principles are less detailed than patterns and do not explain how to achieve the desired goal



# Chess playing vs coding

- Anyone can play chess

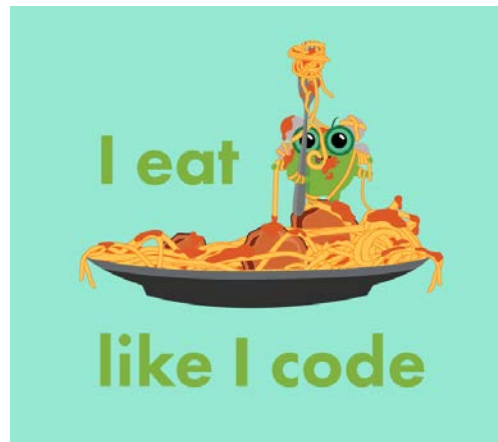


# Chess playing vs coding

- Anyone can play chess



- Anyone can write code ... that compiles



# How can you become a Chess Master?

- Learn rules and physical requirements such as chess board, names of pieces, legal movements, etc.
- Learn principles such as the relative value of each piece, strategic value of center squares, power of a threat, etc.
- To become a chess master, one must study the games of other masters
  - these games contain patterns that must be understood, memorized, and applied repeatedly
  - there are hundreds of these patterns

# How can you become a master Software Engineer?

- Learn the rules such as the algorithms, data structures, programming language, etc.
- Learn the principles such as structured programming, event programming, object oriented programming, etc.
- To become a software engineering artist, you must study the design and implementation of other masters
  - these codes contain patterns that must be understood, memorized and applied repeatedly
  - there are hundreds of these patterns including software design patterns, code smells, architectural patterns, etc.

# Mastering chess vs mastering coding

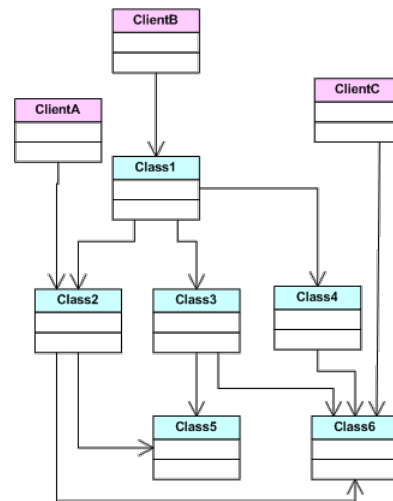
- [Scandinavian opening](#) starts out as

1. e4 d5



and is usually followed by  
2. exd5 Qxd5

- Code



# Mastering chess vs mastering coding

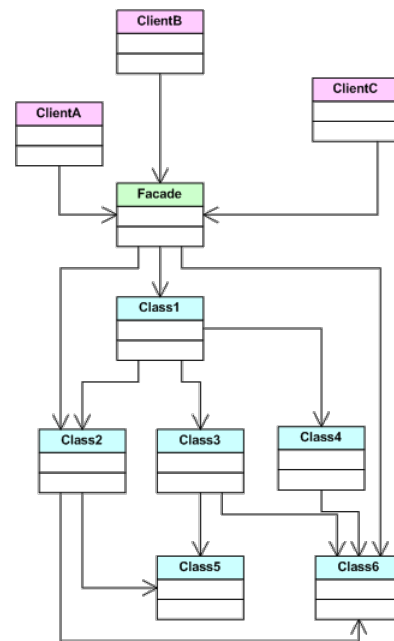
- [Scandinavian opening](#) starts out as

1. e4 d5



and is usually followed by  
2. exd5 Qxd5

- Façade pattern



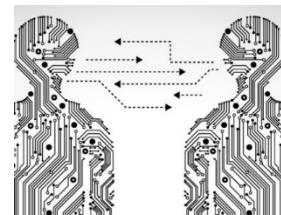
# Design patterns and OOP

- OOP is not mandatory for design patterns
- *“...I think patterns as a whole can help people learn object-oriented thinking: how you can leverage polymorphism, design for composition, delegation, balance responsibilities, and provide pluggable behavior. Patterns go beyond applying objects to some graphical shape example, with a shape class hierarchy and some polymorphic draw method. You really learn about polymorphism when you've understood the patterns. So patterns are good for learning OO and design in general...”*

Erich Gamma et al. “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995

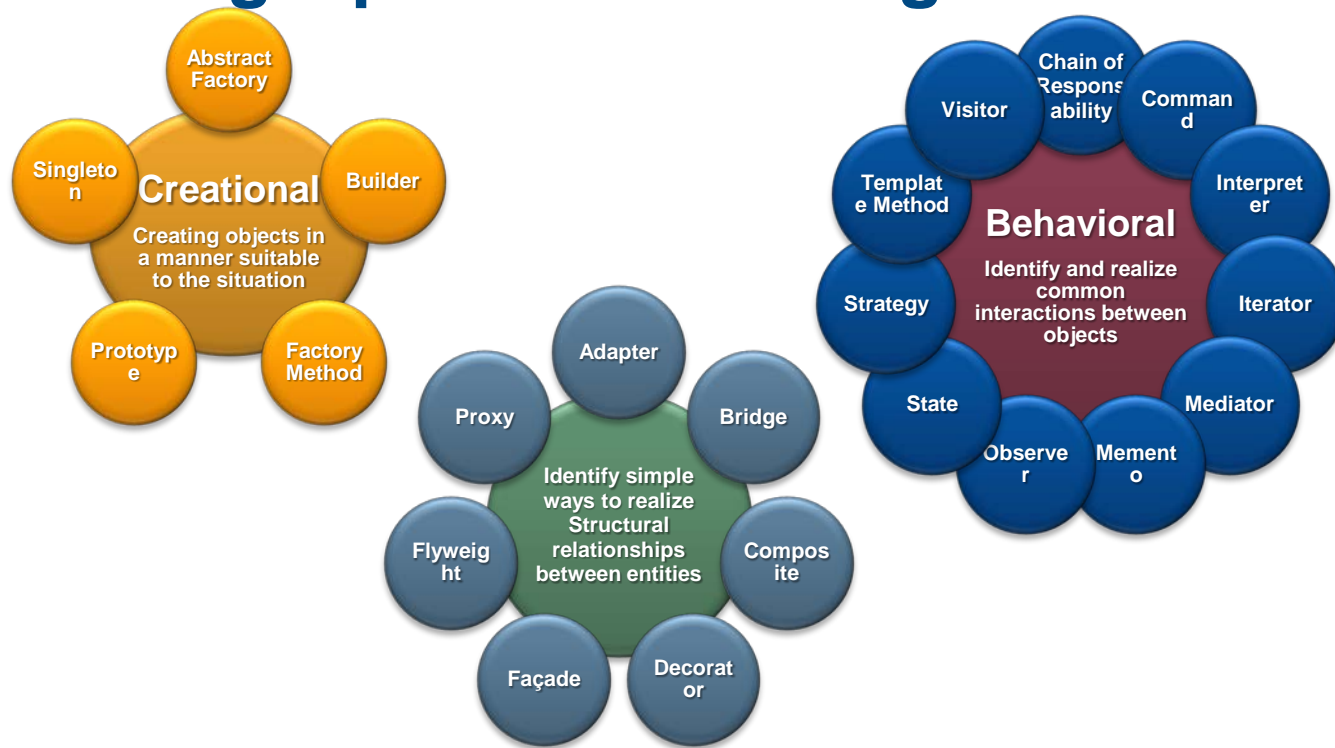
# Types of design patterns

- Creational patterns: used to construct objects in a way that they can be decoupled from the implementing system
  - abstract factory, builder, factory method, prototype, singleton, etc.
- Structural patterns: used to identify simple ways to realize relationships between objects/entities
  - adapter, bridge, composite, decorator, façade, flyweight, proxy, etc.
- Behavioral patterns: used to manage algorithms, relationships and responsibilities between objects
  - chain of responsibility, command, interpreter, iterator, mediator, memento, observer, state, strategy, template method, visitor, etc.

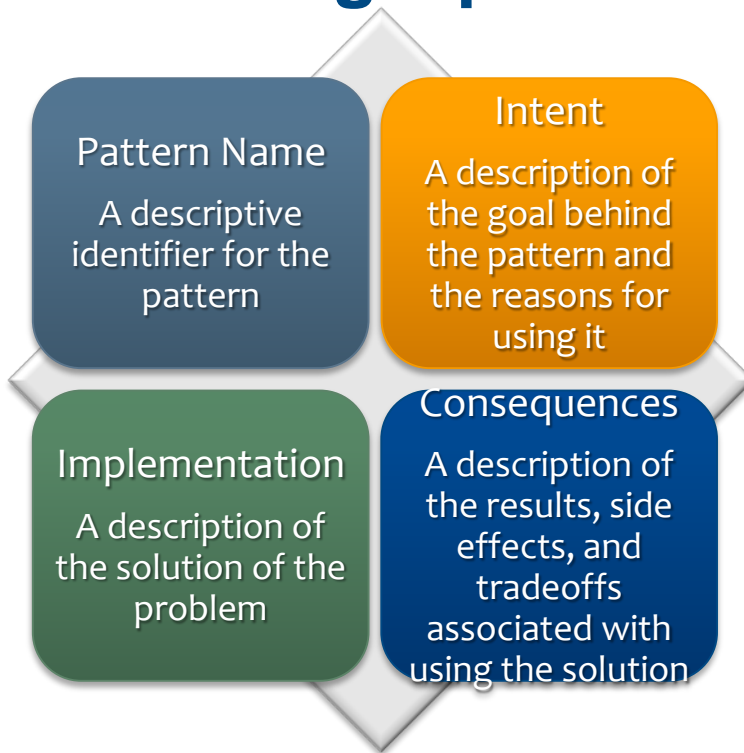




# GoF design patterns: categories



# Description of design patterns



# GoF Design Patterns

## THE 23 GANG OF FOUR DESIGN PATTERNS

<b>C</b> Abstract Factory	<b>S</b> Facade	<b>S</b> Proxy
<b>S</b> Adapter	<b>C</b> Factory Method	<b>B</b> Observer
<b>S</b> Bridge	<b>S</b> Flyweight	<b>C</b> Singleton
<b>C</b> Builder	<b>B</b> Interpreter	<b>B</b> State
<b>B</b> Chain of Responsibility	<b>B</b> Iterator	<b>B</b> Strategy
<b>B</b> Command	<b>B</b> Mediator	<b>B</b> Template Method
<b>S</b> Composite	<b>B</b> Memento	<b>B</b> Visitor
<b>S</b> Decorator	<b>C</b> Prototype	

## The Sacred Elements of the Faith

the holy  
origins

the holy  
structures

107 FM Factory Method							139 A Adapter
117 PT Prototype	127 S Singleton				233 CR Chain of Responsibility	163 CP Composite	173 D Decorator
87 AF Abstract Factory	325 TM Template Method	223 CD Command	273 MD Mediator	293 O Observer	243 IN Interpreter	207 PX Proxy	185 FA Facade
97 BU Builder	313 SR Strategy	283 MM Memento	305 ST State	237 IT Iterator	331 V Visitor	195 FL Flyweight	151 BR Bridge

the holy  
behaviors

Purpose		
Creational	Structural	Behavioral
Abstract Factory Builder Factory Method Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

# GoF Design Patterns

## THE 23 GANG OF FOUR DESIGN PATTERNS

<b>C</b> Abstract Factory	<b>S</b> Facade	<b>S</b> Proxy
<b>S</b> Adapter	<b>C</b> Factory Method	<b>B</b> Observer
<b>S</b> Bridge	<b>S</b> Flyweight	<b>C</b> Singleton
<b>C</b> Builder	<b>B</b> Interpreter	<b>B</b> State
<b>B</b> Chain of Responsibility	<b>B</b> Iterator	<b>B</b> Strategy
<b>B</b> Command	<b>B</b> Mediator	<b>B</b> Template Method
<b>S</b> Composite	<b>B</b> Memento	<b>B</b> Visitor
<b>S</b> Decorator	<b>C</b> Prototype	

## The Sacred Elements of the Faith

the holy  
origins

the holy  
structures

107 FM Factory Method							139 A Adapter
117 PT Prototype	127 S Singleton				233 CR Chain of Responsibility	163 CP Composite	173 D Decorator
87 AF Abstract Factory	325 TM Template Method	223 CD Command	273 MD Mediator	293 O Observer	243 IN Interpreter	207 PX Proxy	185 FA Facade
97 BU Builder	313 SR Strategy	283 MM Memento	305 ST State	237 IT Iterator	331 V Visitor	195 FL Flyweight	151 BR Bridge

the holy  
behaviors

Purpose		
Creational	Structural	Behavioral
Abstract Factory Builder Factory Method Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

## Rationale behind learning design patterns

- You build on your programming skills over a few semesters during your DIBSE education
- When you graduate you learn coding as you practice
  - you will be told over and over again to do so
- By just own practice
  - risk to overlook “collective wisdom” that allows you to build more and more coding skills
  - crystallize your bad coding habits instead of unlearning them

# Refactoring and code smells

- One of the main purposes of design patterns in software engineering is refactoring
- Refactoring is designing the code so internal structure of the programming can be changed without modifying its external behavior
- A code smell is not buggy code, it is a symptom in a program that indicates that refactoring might be needed
  - it is technically correct code that does not prevent the program from functioning yet it may contribute to future technical failure
  - it is a set of coding factors that indicates weaknesses in design that usually results in slowing down development, increasing risk of bugs, increasing difficulty in code maintenance and scalability

# Patterns and anti-patterns

- An anti-pattern is a common reaction to a recurring problem that is gleaned from bad experience
- An anti-pattern usually results in an ineffective and risky solution that is likely to be counterproductive
- Appropriate use of design patterns can help avoid code smells and facilitate refactoring for program improvement

# Design Pattern: Singleton



## Scenario I

- You have been tasked with writing a computer-controlled server for the new web-based coffee machine in your organization
- The machine has an Ethernet port than can be used to send basic commands to the machine such as “place cup into position”, “dispense 1 of 5 types of drinks”, “dispense 1 of 3 additives”, “secure the top to the cup”, etc.
- What are the conditions you have to impose on for the coffee machine controller in this case?

## Scenario II

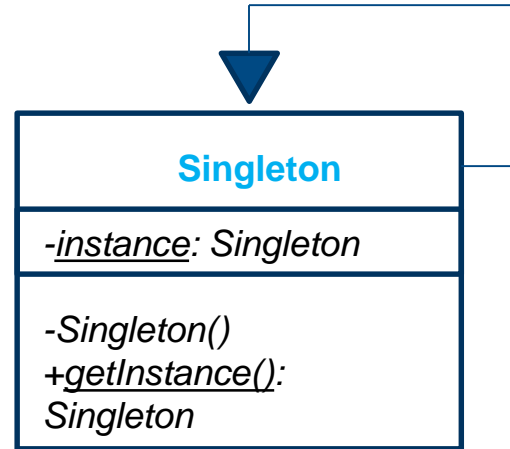
- It is imperative that only one coffee machine controller be running on the server to prevent hodgepodge coffee
- This is an excellent example of the Singleton pattern!

# The singleton pattern

- Creational pattern
- Intent is to
  - ensure that only one instance of a class is created, and
  - to provide a global point of access to that object
- It ensures that the class (not the programmer) is responsible for the number of instances created
- Useful when more than one instance of an object might result e.g. in
  - overuse of resources
  - incorrect behavior and/or inconsistent results
  - need for one global point of access

# Examples and class diagram

- In a system there should be only one window mgr, print spooler, file system because of
  - centralized management of resources
  - provision of a global access point to itself
- Examples to discuss
  - logger
  - counter?
- Typical examples
  - factory
  - application context object
  - thread pool
  - registry setting
  - driver



# Singleton pattern (lazy implementation)

- Implementing a singleton
  - private* constructor prevents object creation from outside
  - a *static* method to get the object
  - creation is done at the latest possible time, only when and if necessary
  - not thread safe

```
public class Singleton {
    private static Singleton instance;

    public static Singleton getInstance() {
        if (instance == null)
            instance = new Singleton();
        return(instance);
    }

    private Singleton() {
        ...
    }
    ...
}
```

# Singleton pattern (lazy thread safe I)

- Making the lazy singleton thread-safe is easy
  - synchronize method `getInstance()`
  - synchronization is expensive, performance can be enhanced

```
public class Singleton {
    private static Singleton instance;

    public synchronized static Singleton getInstance() {
        if (instance == null)
            instance = new Singleton();
        return(instance);
    }

    private Singleton() {
        ...
    }
    ...
}
```

# Singleton pattern (eager implementation I)

- Eager instantiation
  - by declaring reference *C\_INSTANCE* as *static*, the singleton object is instantiated when the class is loaded and not when it is first used

```
public class Singleton {
    private static Singleton C_INSTANCE = new Singleton();

    public static Singleton getInstance() {
        return(Singleton.C_INSTANCE);
    }

    private Singleton() {
        ...
    }


    ...
}
```

# Singleton pattern (eager implementation II)

- Eager instantiation
  - the *final* keyword ensures that no assignment can be made to this reference later
  - *getInstance()* is not necessary since the instance can be accessed as *Singleton.C\_INSTANCE*
  - fine to declare *C\_INSTANCE* as *public* since it is *final*

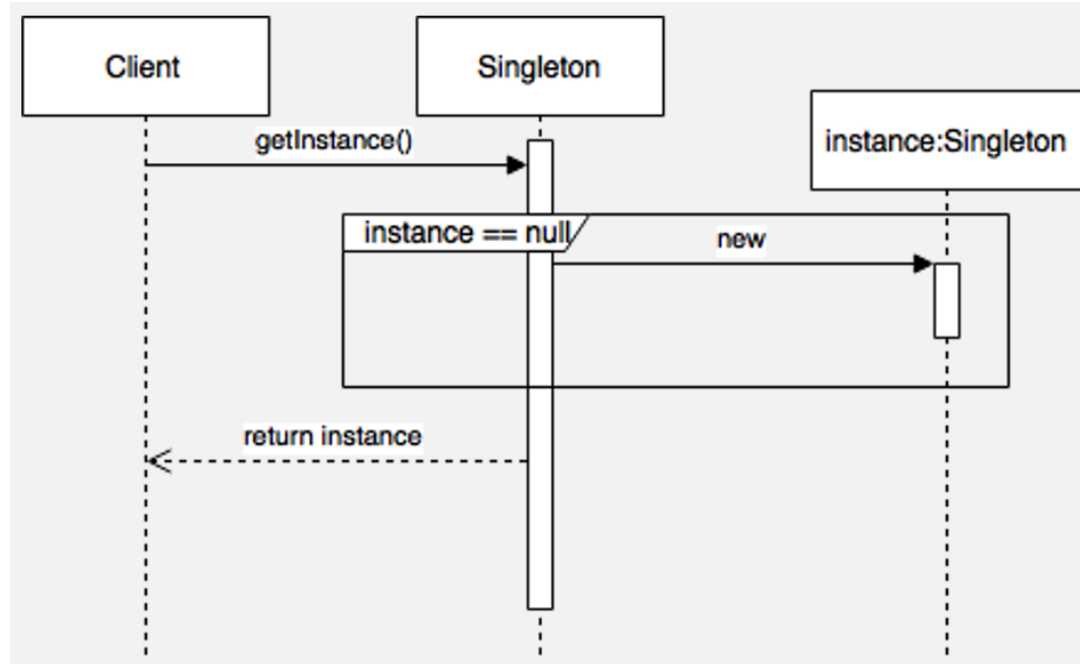
```
public class Singleton {
    public static final Singleton C_INSTANCE = new Singleton();
    public static Singleton getInstance() {
        return C_INSTANCE;
    }

    private Singleton() {
        ...
    }
    ...
}
```





# UML sequence diagram



## How to break the Singleton Pattern in Java

- Java Cloning
- Reflection
- Serialization
- Countermeasures exist for all the above cases

# Singleton with enumeration (I)

- Java makes sure that any enum value is instantiated only once

```
public enum SingletonEnum {
    C_INSTANCE;
    int value;

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }
}
```

## Singleton with enumeration (II)

*“...This approach is functionally equivalent to the public field approach, except that it is more concise, provides the serialization machinery for free, and provides an ironclad guarantee against multiple instantiation, even in the face of sophisticated serialization or reflection attacks. While this approach has yet to be widely adopted, a single-element enum type is the best way to implement a singleton...”*

Joshua Block, Effective Java, Addison.Wesley, 2008

## Known uses

- Examples of the singleton pattern in the JDK
  - *java.lang.Runtime* is a singleton which is returned with *java.lang.Runtime.getRuntime()*
  - *java.awt.Toolkit* is a singleton which is returned with *Toolkit.getDefaultToolkit()*
  - *java.io.Console()* is a singleton which is returned from another class with *java.lang.System.console()*
- Examples of static classes in the JDK
  - *java.lang.Math*
  - *java.lang.System*

# Design Pattern: Factory

# Static factory method

- The factory method pattern is NOT the same as the static factory method
  - static factory method is a static method that returns an instance of a class
  - the idea is to gain control over object creation and delegate it from the constructor to a static method
- The singleton implementation is a special case of static factory method, other common static factory methods:
  - *getInstance()*
  - *newInstance()*
  - *forName()*
  - *valueOf(...)*

# Why using static factory methods

- Does not need to create a new object each time they are invoked, as it would be the case for constructors
  - objects can be cached and can be reused
- Does not need to instantiate an object to call the factory method since it is static
- Can return a subtype of the return type or any implementations of an interface if this is used as the return type of the method
  - return an object whose implementation class is not known to caller



# Static factory method for readability

```
public class Rect {
    public Rect(boolean square) {
        // code here
    }
}
```

What does it mean?

```
Rect r = new Rect(false);
```

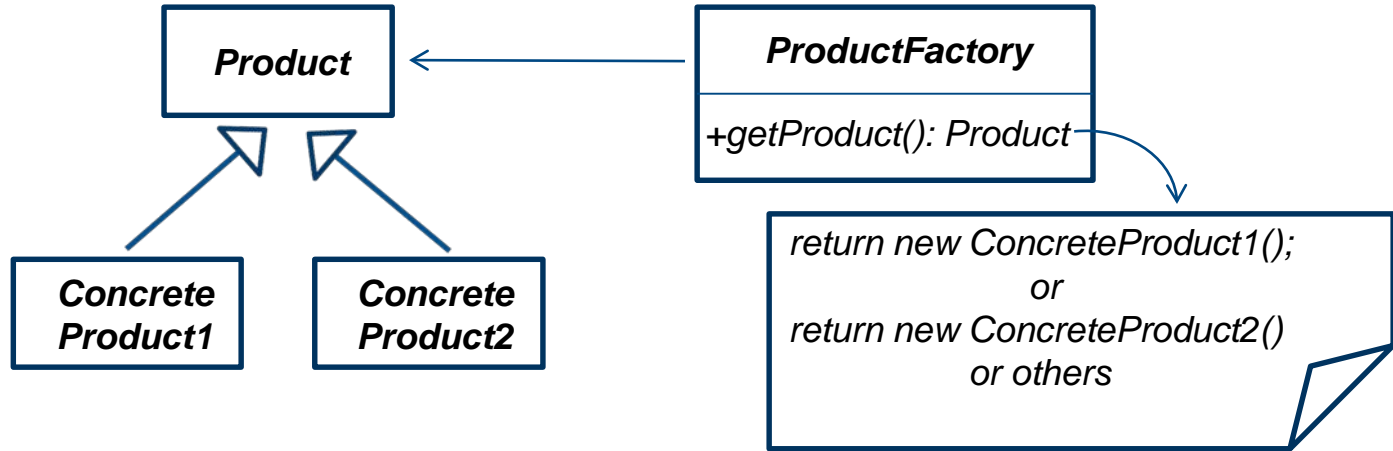
```
public class Rect {
    public static Rect makeSquare() {
        // code here
    }
    ...
    public static Rect makeRect(){
        // code here
    }
}
// much more readable
Rect r = Rect.makeSquare();
```

# Factory method pattern

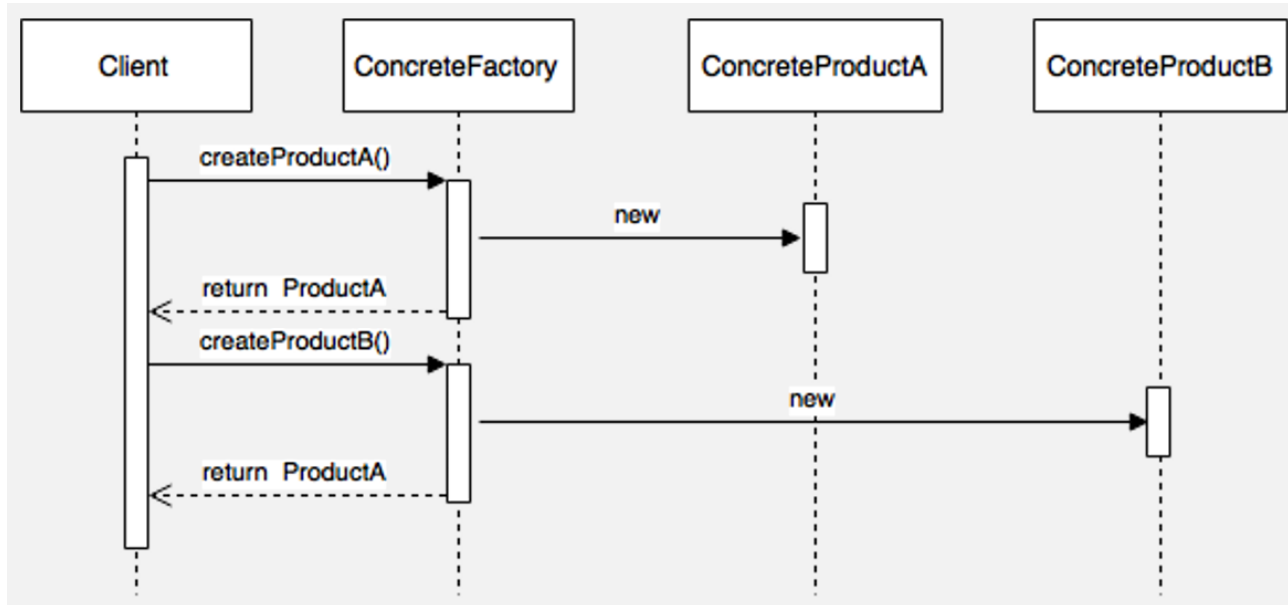


- It is a creational pattern that
  - hides object instantiation, hence lowering code coupling with implementation
  - handles object creation without specifying the exact class of the object that will be created
  - defines an interface/abstract class for creating an object, but let the subclasses decide which actual class to instantiate; it lets a class defer instantiation to subclasses
- It separates responsibility
  - clients need to know how to use an object
  - factory needs to know what object to create exactly
- Factory methods implemented by the factory method pattern
  - are usually not static so that it can be overridden in subclasses
  - constructors of objects being created are either private or protected

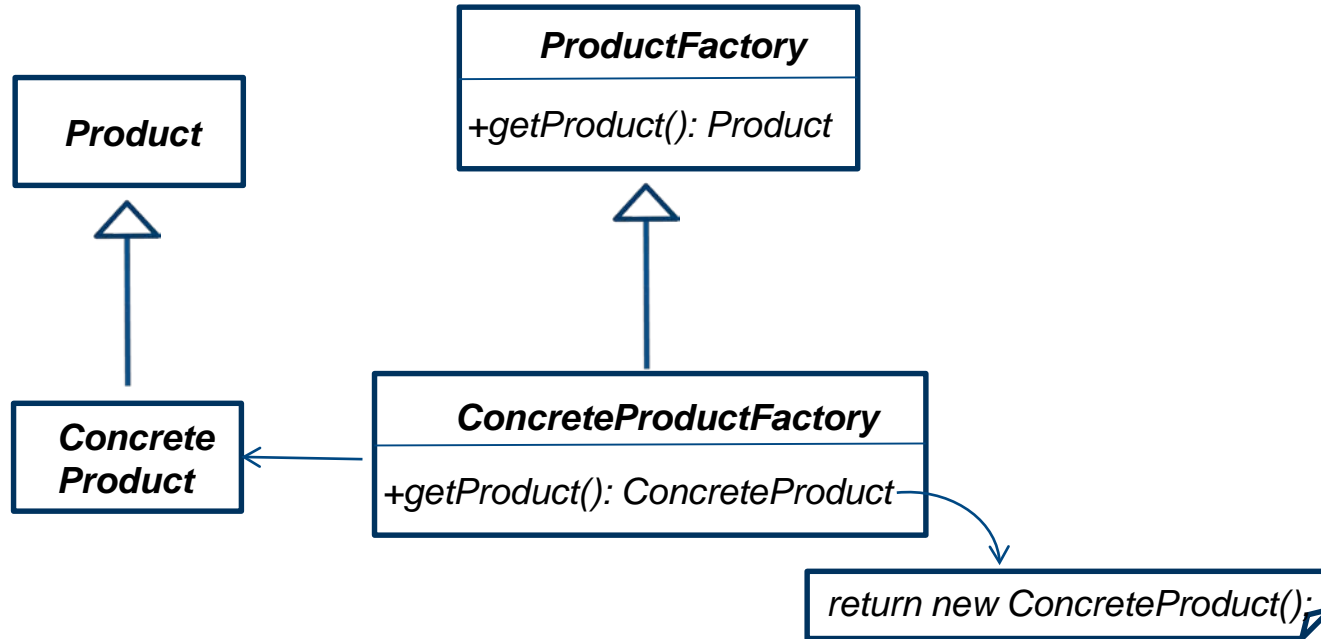
# UML structure diagram (I)



# UML sequence diagram



# UML structure diagram (II)



# Design Pattern: Builder

# Telescoping constructor (anti) pattern

- It is a creational pattern
- It is a pattern to create constructors of a class that has a number of optional parameters
  - a constructor with only the required parameters
  - another constructor with a single optional parameter
  - a third one with two optional parameters and so on
- Many disadvantages
  - scalability
  - readability
  - confusing
  - error prone

# Alternatives to Telescoping constructor

- The JavaBean pattern
  - a default no-arg constructor (or a constructor with the mandatory parameters), setters and getters for every attribute
  - the no-arg constructor is called (or the constructor with the mandatory parameters) and then any optional setters are called
  - not possible with immutable classes. and object may be in an inconsistent state in the midst of its construction
- A set of named static factory methods
  - do not scale well with increasing number of params
- The Builder pattern



# Chaining methods for object creation

- One shot creation

*Product p = new Product(23,"a");*

- Step-by-step object creation with the Builder pattern

*Builder builder = new Product.Builder();*

*Product p = builder.createPart1(23).createPart2("a");*

- Added readability

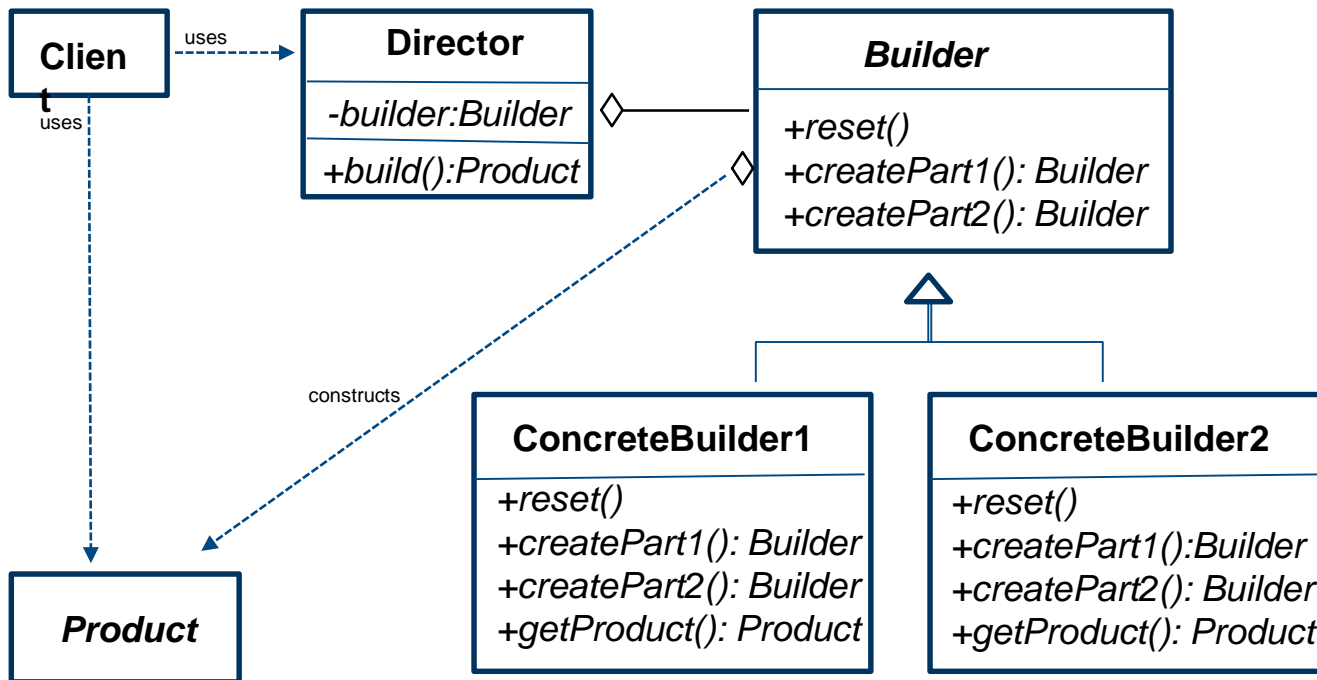
*Rectangle r = new Rectangle(10,20);*                      vs

*Rectangle r = builder.width(10).height(20);*

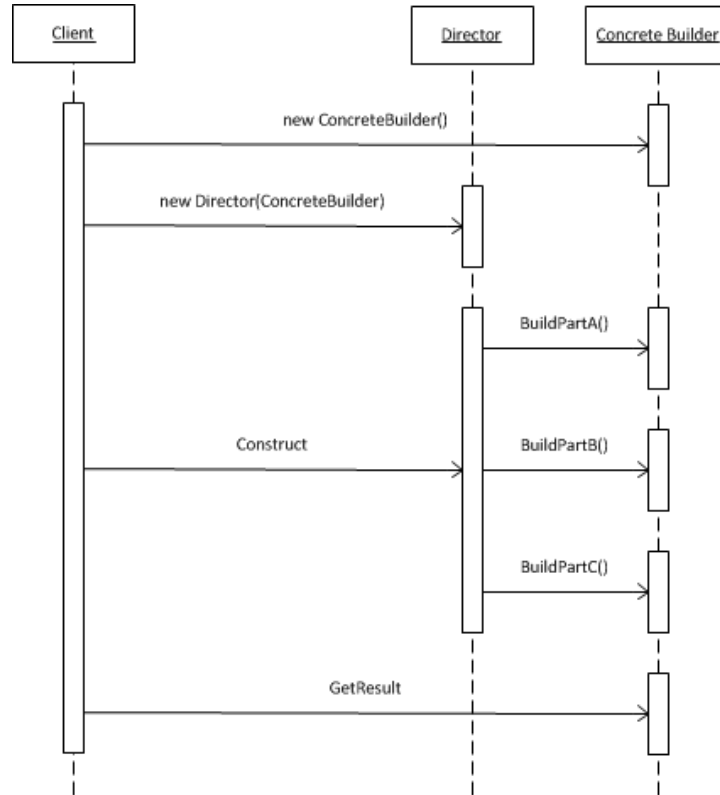
# Builder pattern

- It is a creational pattern
- It separates the construction of a complex object from its representation
  - creates complex objects by creating constituent parts
  - the same construction process may be utilized to create different representations of the object
  - builds the product as a configurable step by step sequence instead of in one shot
  - client code has control over the object creation process
- New creation algorithm can be added without changing code

# UML structure diagram



# UML sequence diagram



# Static nested builder class

- A static nested class *Builder* is defined inside the class *Product* whose object will be build by *Builder*
- *Builder* has the same set of fields as original class *Product*
- *Builder* exposes methods for setting attributes; each method returns the same *Builder* object
  - *Builder* is enriched with each method call
- A method *Builder.build()* is defined to copy all builder field values into actual class and return object of original class *Product*
- *Product* has private constructor to create its object from *build()* method and prevent outsider to access its constructor

# Builder pattern: pros and cons

- Code is more maintainable if number of fields required to create object is large (4+)
- Object creation code less error-prone as users know what they are passing because of explicit call
- Provides finer control over the step-by-step construction process when the object being produced is complex
- Concentrates on *how* it is being built, while the other creational patterns focus on *what* is being built
- Provides a way to build immutable objects
- It is verbose and requires code duplication
- Known uses: *StringBuilder*, *Stream.Builder*, *Locale.Builder*, etc.

# Weekly assignments: deadlines reminder

- Assignment 3: Thursday, November 14 at 23:55

# SOFTWARE ENGINEERING I

## Design Patterns II

[andrea.corradini@mci.edu](mailto:andrea.corradini@mci.edu)





## Tentative topics

- Webinar 1: NetBeans IDE and Maven, Java language I: Java objects, classes, and interfaces
- Webinar 2: Java language II: Java inheritance, polymorphism, and exception handling
- Webinar 3: design patterns I: selected creational patterns
- Webinar 4: design patterns II: selected structural patterns
- Webinar 5: design patterns III
- Webinar 6: OOP principles

# GoF Design Patterns

THE 23 GANG OF FOUR DESIGN PATTERNS

<b>C</b> Abstract Factory	<b>S</b> Facade	<b>S</b> Proxy
<b>S</b> Adapter	<b>C</b> Factory Method	<b>B</b> Observer
<b>S</b> Bridge	<b>S</b> Flyweight	<b>C</b> Singleton
<b>C</b> Builder	<b>B</b> Interpreter	<b>B</b> State
<b>B</b> Chain of Responsibility	<b>B</b> Iterator	<b>B</b> Strategy
<b>B</b> Command	<b>B</b> Mediator	<b>B</b> Template Method
<b>S</b> Composite	<b>B</b> Memento	<b>B</b> Visitor
<b>S</b> Decorator	<b>C</b> Prototype	

The Sacred Elements of the Faith

the holy  
origins

the holy  
structures

107 FM Factory Method							139 A Adapter
117 PT Prototype	127 S Singleton				233 CR Chain of Responsibility	163 CP Composite	173 D Decorator
87 AF Abstract Factory	325 TM Template Method	223 CD Command	273 MD Mediator	293 O Observer	243 IN Interpreter	207 PX Proxy	185 FA Facade
97 BU Builder	313 SR Strategy	283 MM Memento	305 ST State	237 IT Iterator	331 V Visitor	195 FL Flyweight	151 BR Bridge

the holy  
behaviors

Purpose		
Creational	Structural	Behavioral
Abstract Factory Builder Factory Method Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

# Design Pattern: Composite

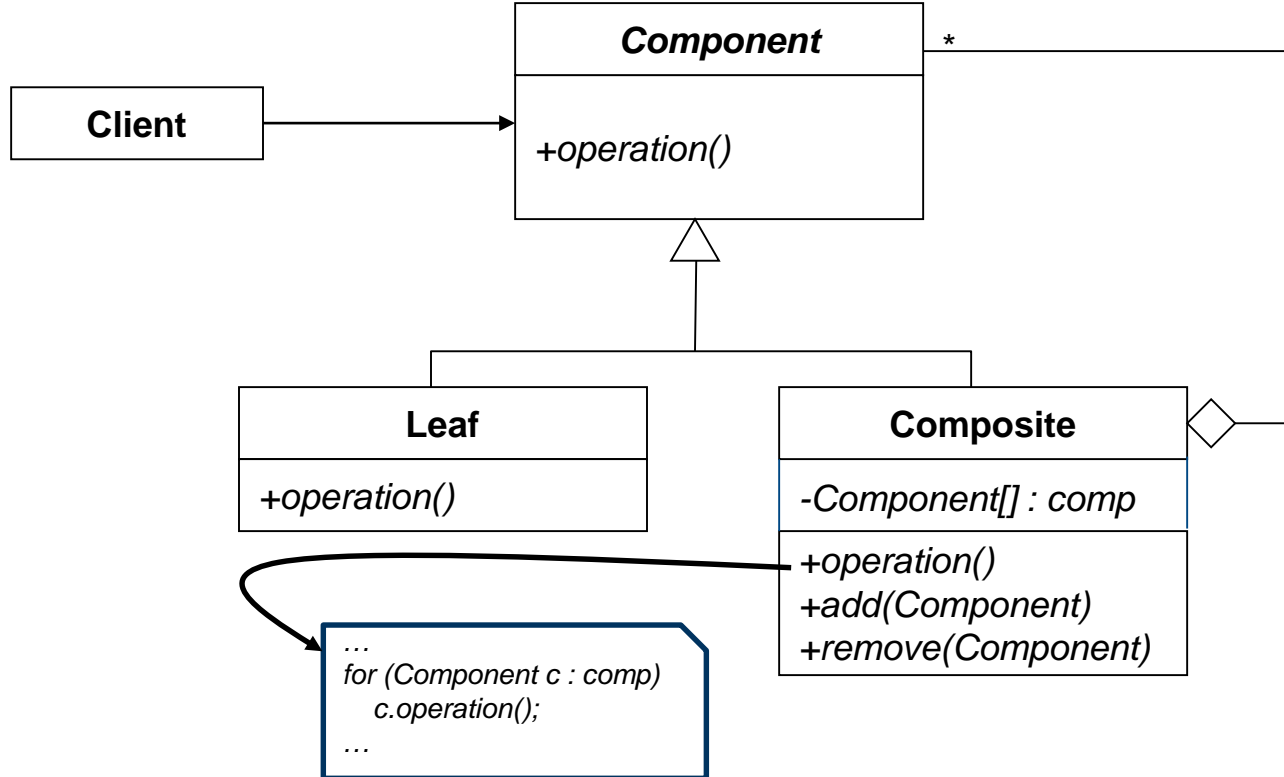
# The composite pattern

- It is a structural pattern
- It allows to compose objects into tree structures to represent part-whole hierarchies and lets clients treat individual objects and compositions of objects uniformly
- It defines class hierarchies consisting of primitive objects and composite objects
  - primitive objects can be composed into more complex objects, which in turn can be composed
- Every element in the structure (primitive and composite objects) operates with a uniform interface
  - adding new components is easy and client code remains unchanged

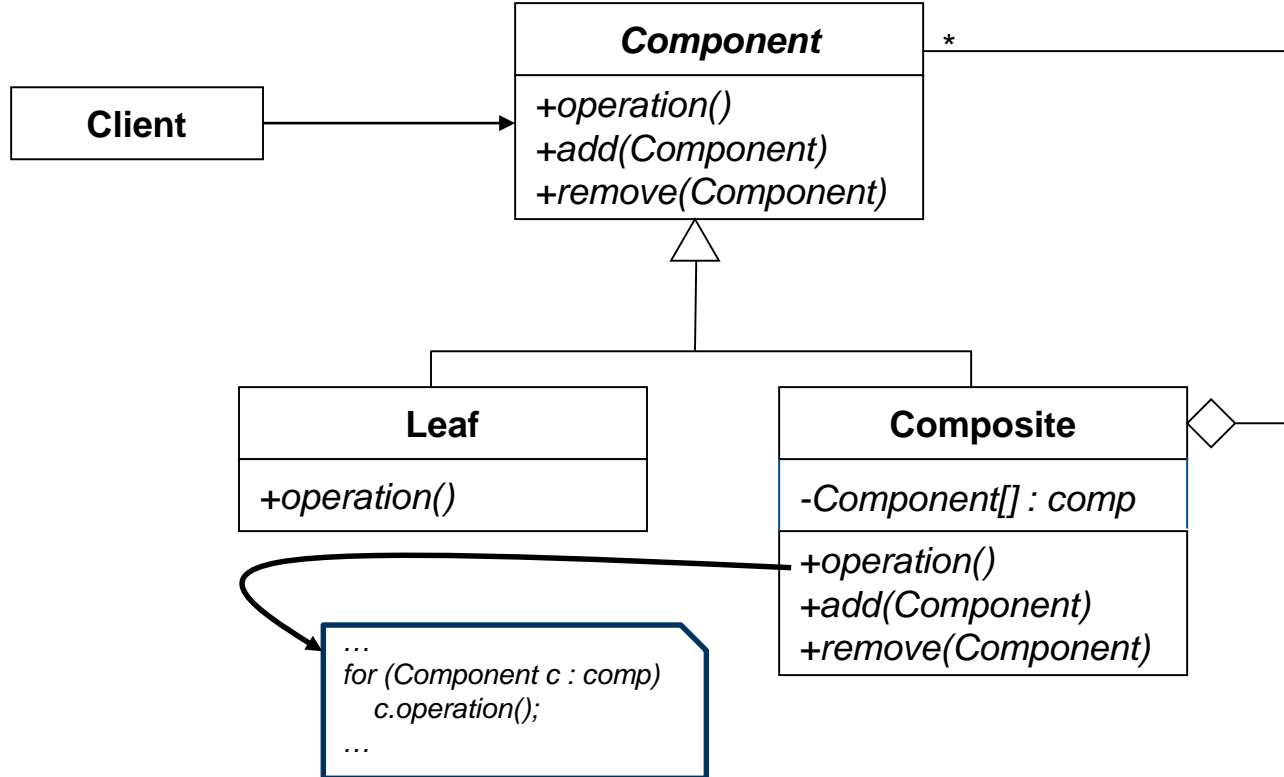
# Composite objects

- A composite is an object composed of either individual items or other composites or a mix thereof
  - composites are objects that can hold themselves
- Example: a drawing is a composite being composed of graphic primitives, like lines, rectangles, circles, etc.
  - each graphic primitive can be drawn, moved, and resized; the same operations can also be performed on the composite
  - manipulation of both primitive objects and composite objects in exactly the same way, i.e. without distinguishing between them, simplifies code implementation and maintenance

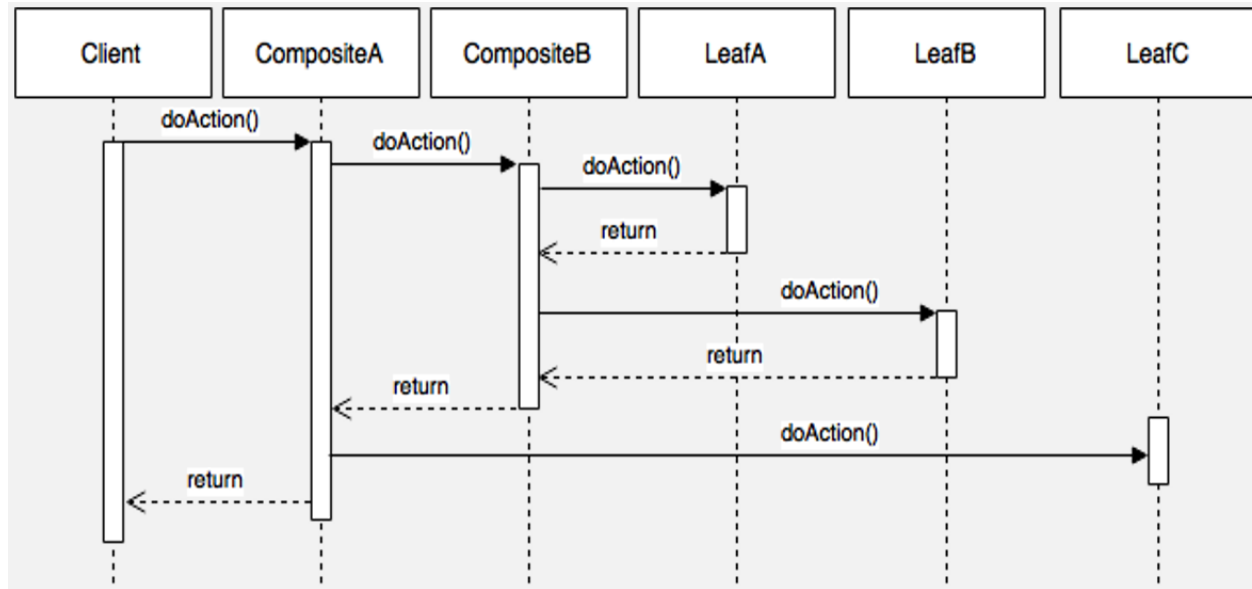
# UML structure diagram



# UML structure diagram



# UML sequence diagram





## Known uses

- At the heart of the pattern is the ability for a client to perform operations on an object without needing to know that there are many objects inside
- The composite design pattern lets you treat primitive and composite objects exactly the same way; this also happens in other frameworks and domains
  - the Apache Struts framework includes Tiles, a JSP tag library, that allows to compose a Web page from multiple JSPs
  - the Java GUI layout and widgets
  - the JUnit 3.x is a testing framework for Java ([www.junit.org](http://www.junit.org))

# Design Pattern: Decorator

# The Decorator Pattern

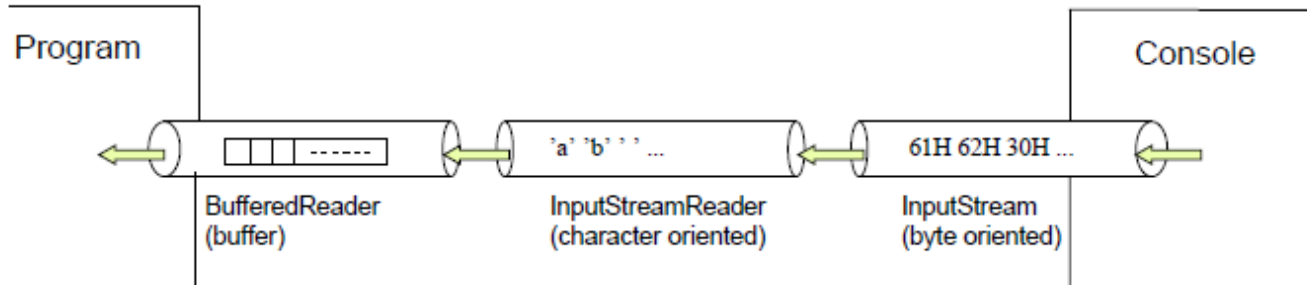
- It is a structural pattern
- When objects are alike in some fundamental way, but might have a variety of distinctive details, this pattern avoids the complexity of having a large number of derived classes
- The decorator is an object that modifies the behavior of, or adds features to another object
  - decorator must maintain the interface of the object it wraps up
  - features can be added to an object without disrupting the interface

# Example in Java core library (I)

- Multilayered input streams adding useful I/O methods

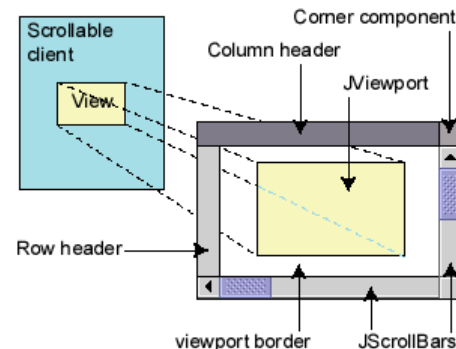
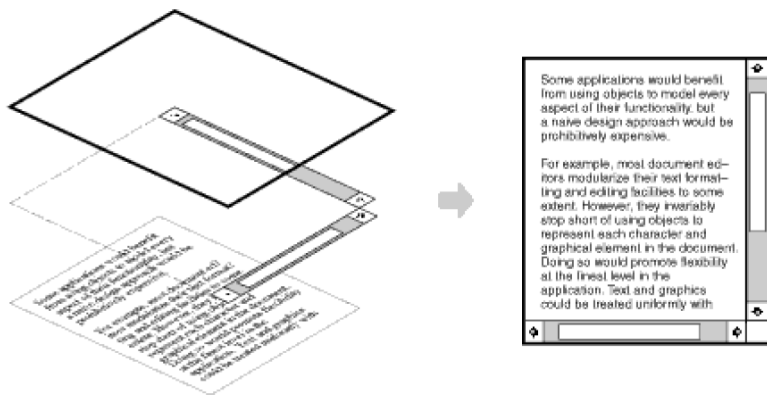
Input from console:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in))
```

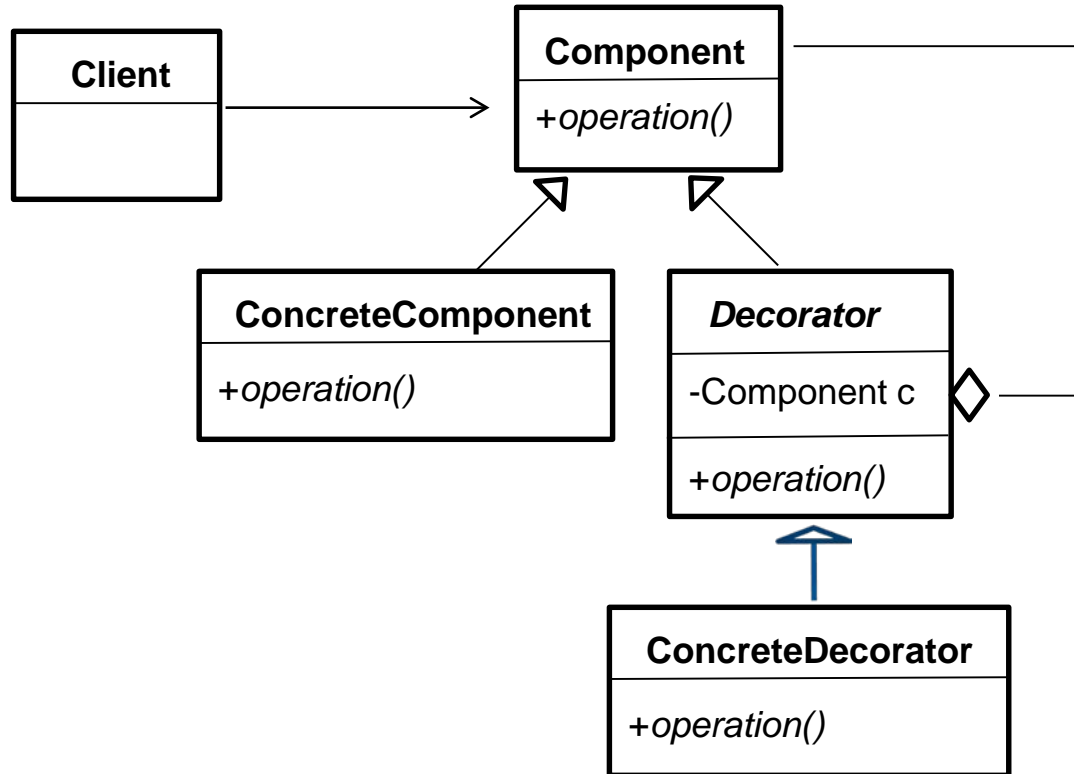


## Example in Java core library (II)

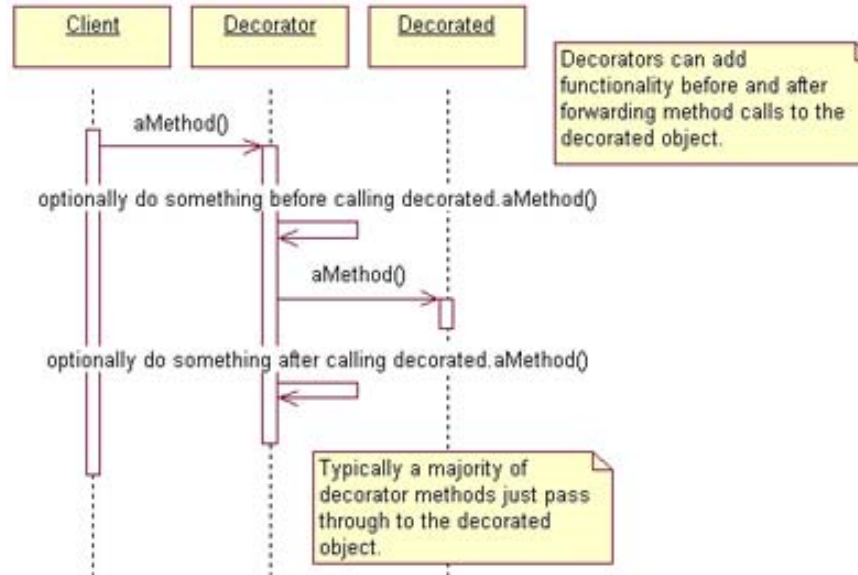
- Adding designs, scroll bars and borders to GUI controls
  - normal GUI components do not have scroll bars
  - *JScrollPane* is a container with scroll bars to which you can add any component to make it scrollable



# UML structure diagram



# UML sequence diagram



# Common usage

- When to use the decorator patterns:
  - concrete implementations are decoupled from behaviors
  - subclassing to achieve modification is impractical or impossible
  - specific functionality should not reside high in the object hierarchy
  - many little objects surrounding a concrete implementation are acceptable



# Pros and cons

- Pros
  - provides a more flexible way to add responsibilities to a class than the use of inheritance
  - allows to customize a class without creating subclasses high in the inheritance hierarchy
  - achieves complex behaviors by combining rather simple classes
- Cons
  - understand the concept of putting functionality together
  - know combinations and order of functionality that suit a need
  - more lines of code to accomplish common I/O tasks
  - code maintenance difficult because of many similar looking objects

# Design Pattern: Proxy

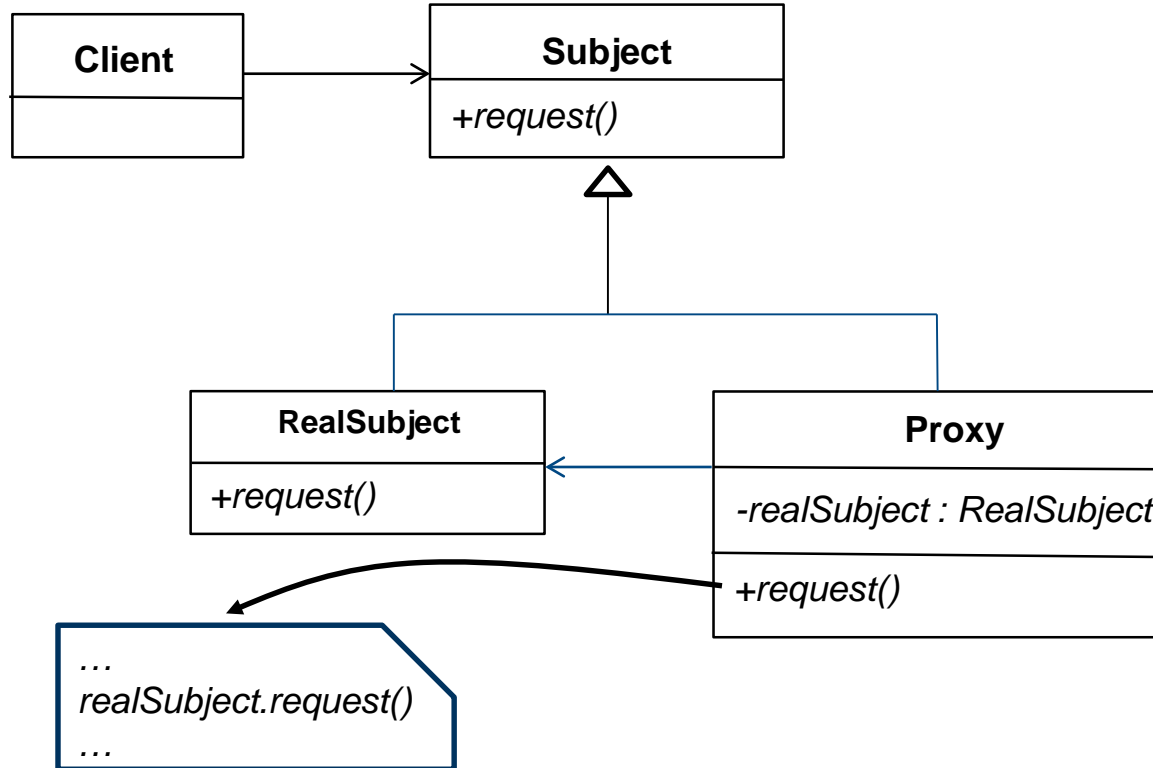
# The Proxy Pattern

- It is a structural pattern
- It provides a surrogate for another object, for reasons such as access, speed, or security
  - at times, a client needs to interact with an object without accessing it directly
- It is applicable whenever there is a need for a more versatile reference to an object than a regular pointer
- It provides an interface identical to the subjects, controls access to the real subject and also may be responsible for creating and deleting it

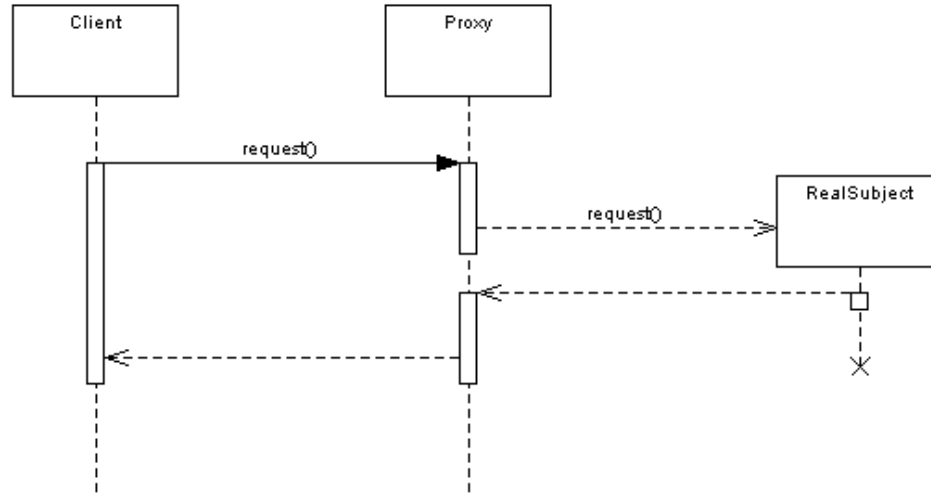
# Proxy types

- Remote proxy: provides a local representative for object in a different address space
- Virtual proxy: creates objects on demand e.g. for optimizations
- Protection proxy: controls access to original object when access rights differ
- Smart reference: replaces bare pointer for counting references, deleting when no longer referenced, etc.
- Firewall proxy: protects targets from bad clients
- Cache proxy: provides temporary storage of expensive target operations so multiple clients can share the results
- Synchronization proxy: provides multiple accesses to a target object

# UML structure diagram

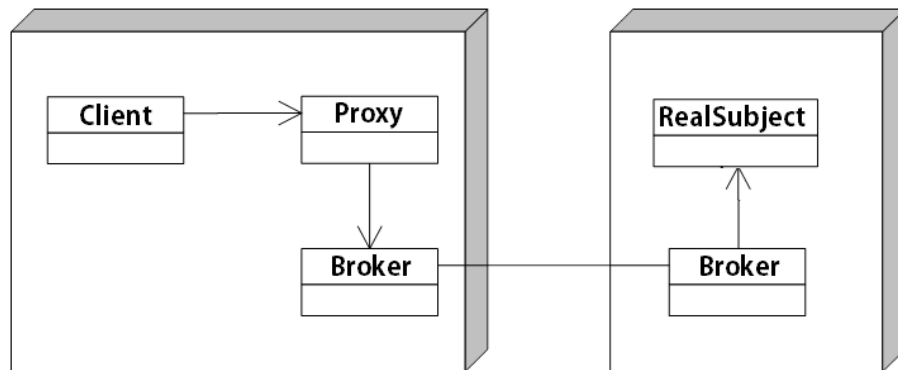


# UML sequence diagram



# Known uses and related patterns

- Distributed objects



- Related patterns:
  - adapter
  - decorator

# SOFTWARE ENGINEERING I

## Design Patterns III

[andrea.corradini@mci.edu](mailto:andrea.corradini@mci.edu)





# Tentative topics

- Webinar 1: NetBeans IDE and Maven, Java language I: Java objects, classes, and interfaces
- Webinar 2: Java language II: Java inheritance, polymorphism, and exception handling
- Webinar 3: design patterns I: selected creational patterns
- Webinar 4: design patterns II: selected structural patterns
- Webinar 5: design patterns III: selected behavioral patterns
- Webinar 6: OOP principles

# GoF Design Patterns

THE 23 GANG OF FOUR DESIGN PATTERNS

<b>C</b> Abstract Factory	<b>S</b> Facade	<b>S</b> Proxy
<b>S</b> Adapter	<b>C</b> Factory Method	<b>B</b> Observer
<b>S</b> Bridge	<b>S</b> Flyweight	<b>C</b> Singleton
<b>C</b> Builder	<b>B</b> Interpreter	<b>B</b> State
<b>B</b> Chain of Responsibility	<b>B</b> Iterator	<b>B</b> Strategy
<b>B</b> Command	<b>B</b> Mediator	<b>B</b> Template Method
<b>S</b> Composite	<b>B</b> Memento	<b>B</b> Visitor
<b>S</b> Decorator	<b>C</b> Prototype	

The Sacred Elements of the Faith

the holy  
origins

the holy  
structures

107 FM Factory Method							139 A Adapter
117 PT Prototype	127 S Singleton				233 CR Chain of Responsibility	163 CP Composite	173 D Decorator
87 AF Abstract Factory	325 TM Template Method	223 CD Command	273 MD Mediator	293 O Observer	243 IN Interpreter	207 PX Proxy	185 FA Facade
97 BU Builder	313 SR Strategy	283 MM Memento	305 ST State	237 IT Iterator	331 V Visitor	195 FL Flyweight	151 BR Bridge

the holy  
behaviors

Purpose		
Creational	Structural	Behavioral
Abstract Factory Builder Factory Method Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

# Design Pattern: Template Method

# Example: Java on different platforms (I)

- Algorithm for running Java SE applications
  - load Java source code
  - parse Java source code
  - compile source into *.class* bytecode
  - feed bytecode to Java VM
- Algorithm for running Java Card applications
  - load Java source code
  - parse Java code
  - compile source into *.class* bytecode
  - feed bytecode to Java Card VM
- Algorithm for running Android applications
  - load Java source code
  - parse Java code
  - compile source into *.dex* bytecode
  - feed bytecode to Android runtime (ART)

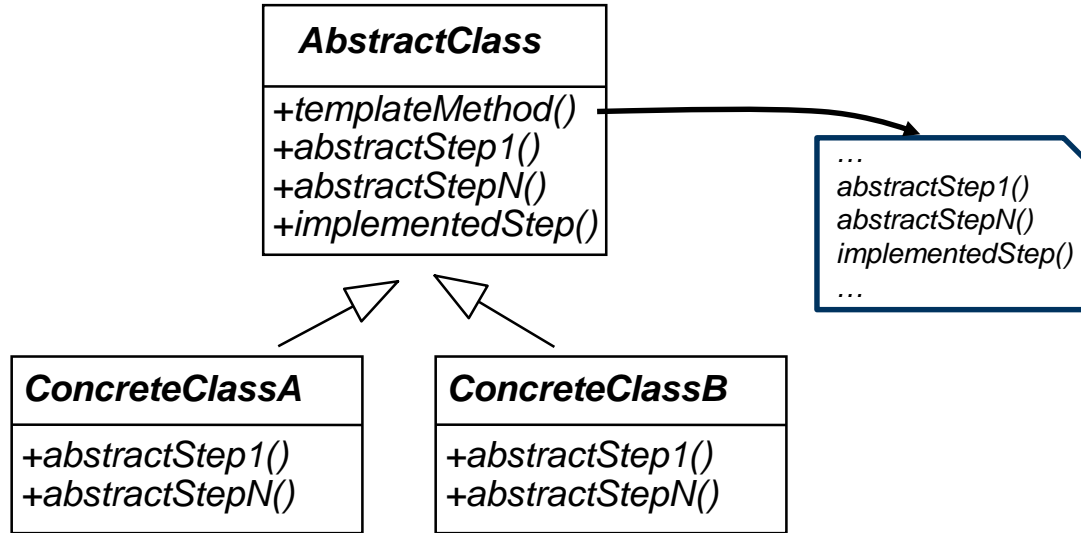
*A common algorithm*

-----  
*load()*  
*parse()*  
*compile()*  
*execute()*

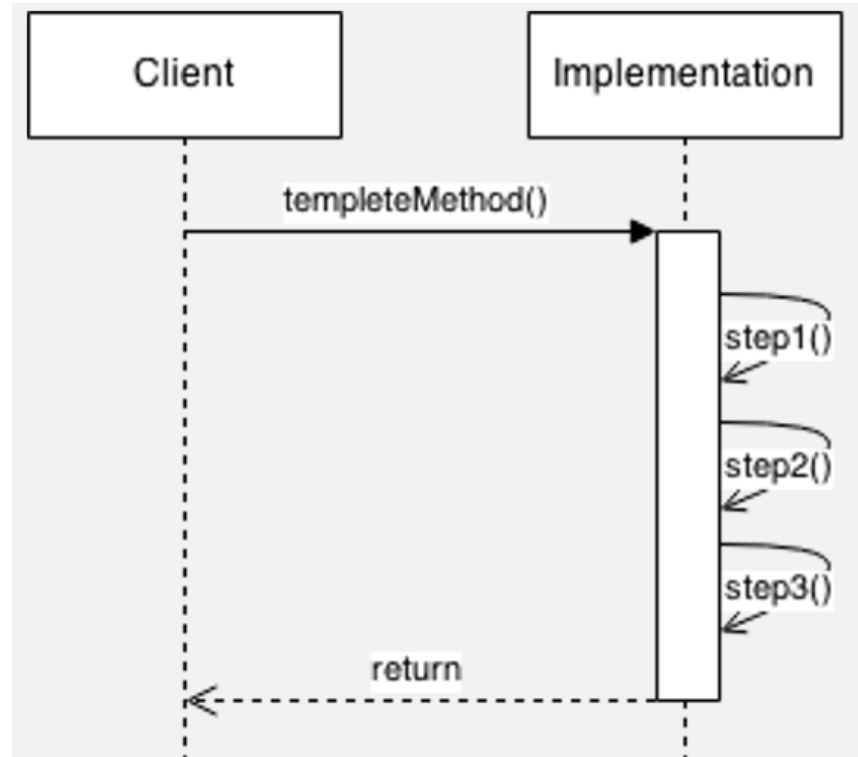
# The template method pattern

- It is a behavioral pattern
- It defines the skeleton of an algorithm as sequence of methods and defers some steps for subclasses to implement
  - the skeleton (i.e. the actual template method) is defined in a base class, with placeholders replacing certain key steps
  - certain common steps are defined and implemented in the base class
  - the specialized steps used in the template method are declared as abstract and overridden by subclasses to allow different behaviors

# UML structure diagram



# Sequence diagram



# Example: Java on different platforms (II)

```
public abstract class RunJava {
    public final void execute() {
        load();
        if (parse()) {
            compile();
        }
    }
    public abstract void compile();
    public void load() {
        // concrete implementation
    }
    public void parse() {
        // concrete implementation
    }
}
```

```
public class RunJavaSE extends RunJava {
    public void compile() {
        // Java SE's concrete implementation
    }
}

public class RunJavaCard extends RunJava {
    public void compile() {
        // Java Card's concrete implementation
    }
}

public class RunAndroid extends RunJava {
    public void compile() {
        // Android's concrete implementation
    }
}
```



# Examples in the JDK

- *java.util.Collections* and *java.util.Arrays* provide a sorting algorithm as a template method that defers the comparison step to the coder
- This method sorts the specified array of objects according to the order induced by the given comparator  
`static <T> void sort(T[] a, Comparator<? super T> c)`
- This method sorts the list passed as parameter according to the order induced by a comparator object:  
`static <T> void sort(List<T> list, Comparator<? super T> c)`

# Template method hooks

- The number of abstract methods being used should be minimized
  - the steps of the algorithm should not be too granular
  - less granularity implies less flexibility
- A hook is a method that a coder places in his code to give others a chance to insert code at a specific location
  - usually to implement an optional part of an algorithm
- Hooks are often used in the Java API
  - *public void paint(Graphics g)* and *public void repaint()* in *java.awt.Component*
  - *void init()*, *void start()*, *void stop()*, *void destroy()* in *java.applet.Applet*

# The Hollywood principle

- The template pattern uses the Hollywood principle
  - “don’t call us, we’ll call you”
- The abstract class is the high-level component that controls the overall process
  - low-level components, i.e. concrete classes, are activated by high-level components (the abstract classes)
  - low-level components never call a high-level component i.e. do not depend on a high level component

## Pros and cons

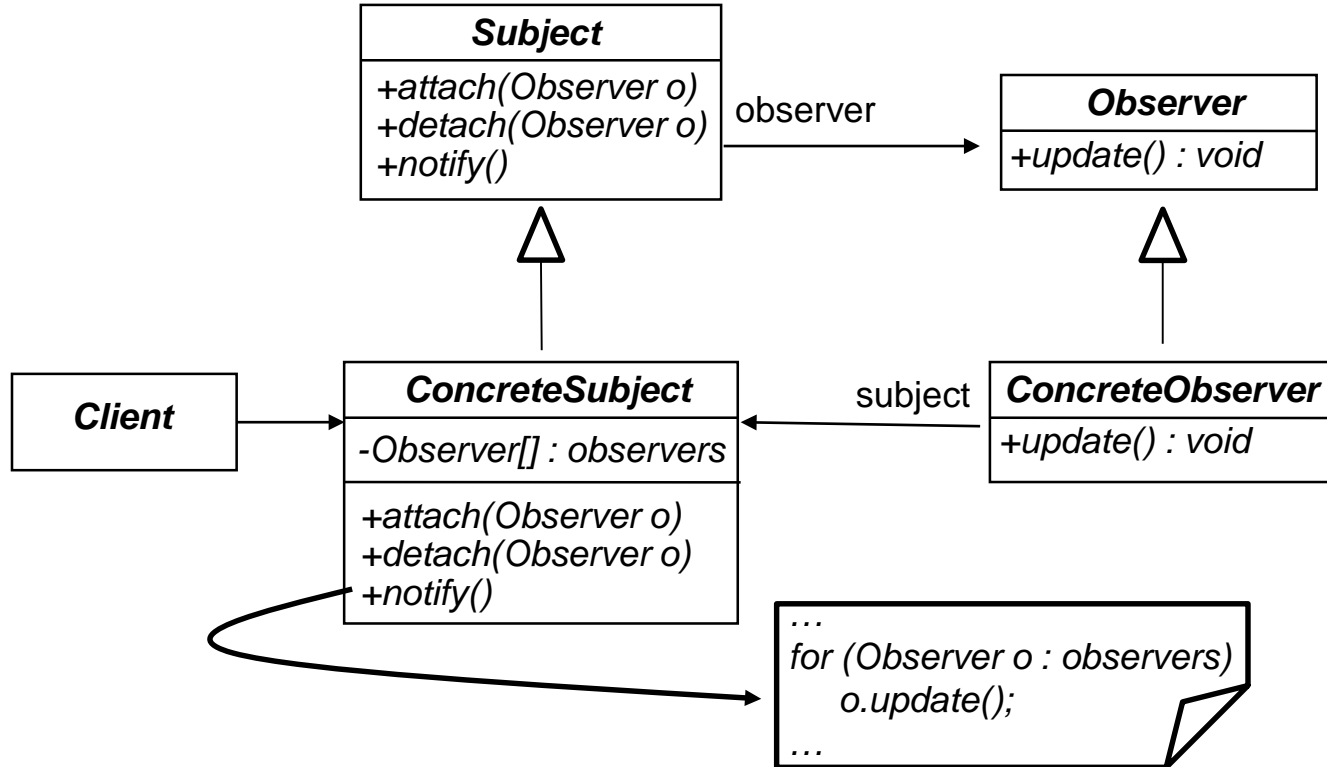
- The primary advantage is the elimination of code duplication
- In general, despite composition is favored over inheritance, this pattern favors inheritance, since it achieves greater flexibility by allowing subclasses to use some operations of the superclass while overriding others
- Takes advantage of polymorphism, allowing the base class to automatically call the methods of the correct subclasses

# Design Pattern: Observer

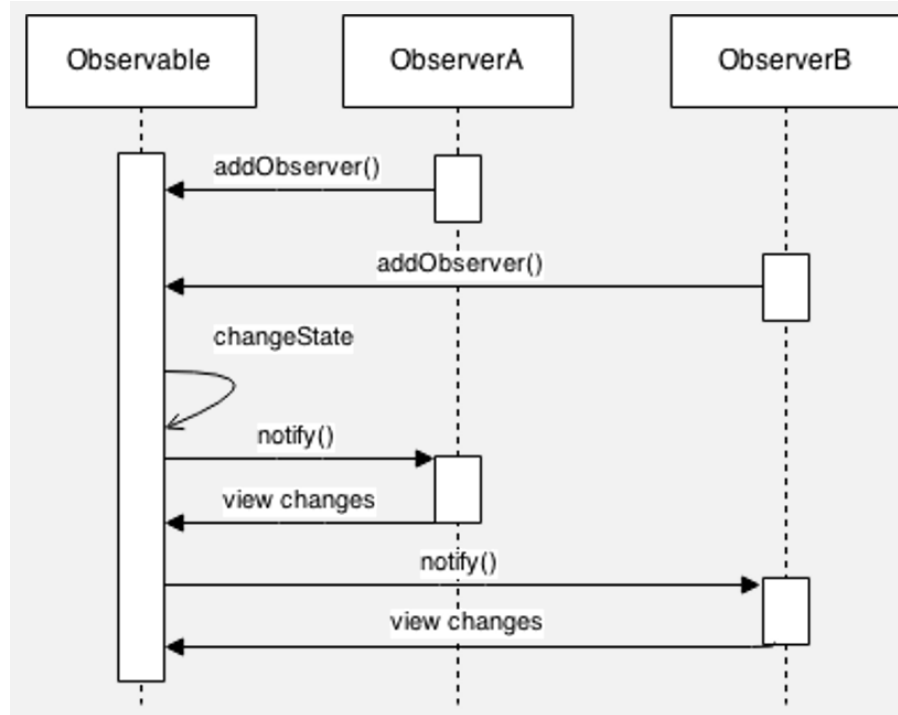
# The Observer Pattern

- It is a behavioral pattern
- The observer pattern implements a one-to-many relationship at run-time
  - a single object, the subject, can change state
  - one or more objects, the observers, are interested in the state of the subject and register their interest with it
  - when something changes in the subject, a notify message is sent to the observers
- It supports decoupling among objects

# UML structure diagram



# UML sequence diagram





## Pros and cons

- Subject and observers are loosely coupled
- Subject and observer can be reused separately and vary independently
- Support for broadcast communication
- A large monolithic design does not scale well as additional graphical and monitoring requirements are added
- Difficult to follow the control flow
- Memory management necessary
- Updates might be costly

## Known uses

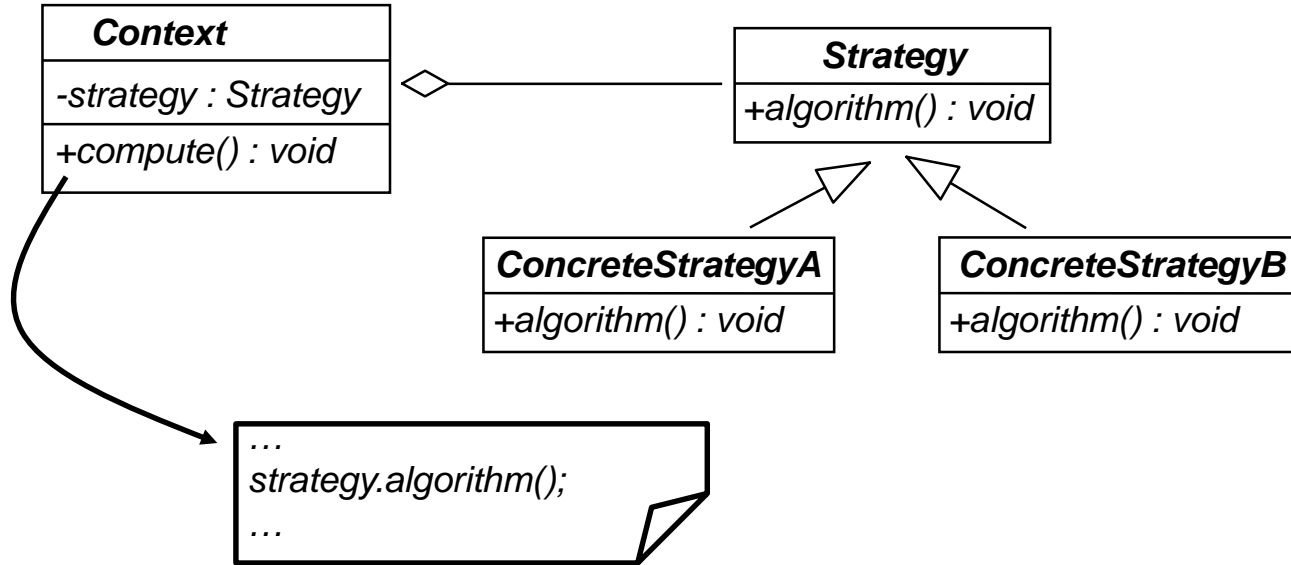
- Swing/GUI and listeners
- Java Message Service
- MVC framework

# Design Pattern: Strategy

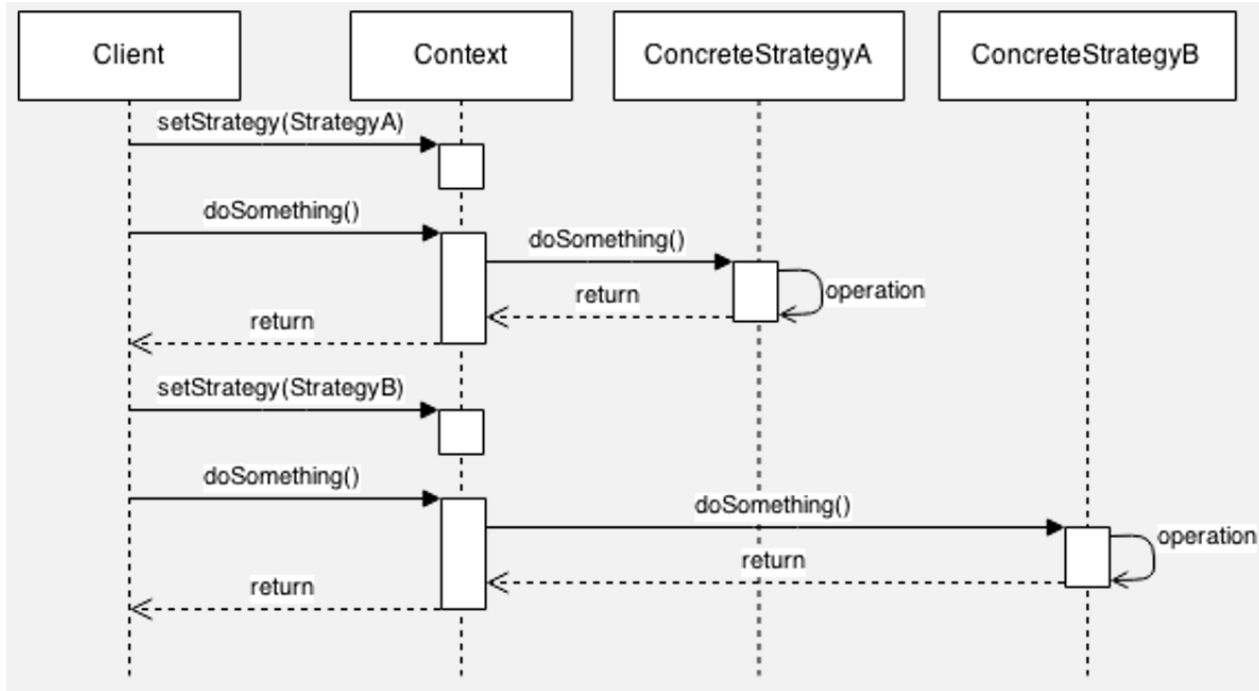
# The Strategy Pattern

- It is a behavioral pattern
- It defines a group of classes that represent a set of possible behaviors that can be plugged into an application, changing the functionality on the fly
- It allows changing the behavior of an object dynamically without changing the object itself
- Strategies provide a way to configure one class with many behaviors when
  - related classes differ only in their behavior
  - an algorithm must be selected at run-time
  - want to add to the possible ways to perform an action

# UML structure diagram



# UML sequence diagram



# Pros and cons

- It separates behaviors from the object that wants to perform it
- It eliminates large conditional statements
- It easier to keep track of different behavior because they are in different classes
- It increases the number of objects

# SOFTWARE ENGINEERING I

## OOP Principles

[andrea.corradini@mci.edu](mailto:andrea.corradini@mci.edu)





# Tentative topics

- Webinar 1: NetBeans IDE and Maven, Java language I: Java objects, classes, and interfaces
- Webinar 2: Java language II: Java inheritance, polymorphism, and exception handling
- Webinar 3: design patterns I: selected creational patterns
- Webinar 4: design patterns II: selected structural patterns
- Webinar 5: design patterns III: selected behavioral patterns
- Webinar 6: OOP principles

# OOP Principles

- A number of rules of thumb about OO design forms design principles
- Guidelines to abide by for the design to end up being a good one, easy to implement and to maintain
  - some overlap
  - one or more of the design principles addressed may depend on others

Principle:

*Encapsulate things in your  
design that are likely to  
change*

# Data access

- Encapsulation separates the contractual interface of an abstraction from its implementation
  - is a mechanism used to hide the data, internal structure, and implementation details of an object
  - all interaction with objects is through a public interface of operations
  - separation of class implementation from use of the class (i.e. the class interface)
  - modularity of application components
- Classes should not expose their internal implementation details
- Minimize the accessibility of classes and members
  - abstraction is a fundamental way to deal with complexity
  - abstraction focuses on the outside view of an object and separates an its behavior from its implementation

# Enforcing encapsulation

- Do not use *public* data members, instead use accessors and mutators to access data members
  - if users of your class accessed the fields directly, then they would be responsible for checking constraints
  - constraints on data members should take place in accessors (getters and setters)
  - the internal processing can be changed without changing the interface
    - setters/getters ensures that only legal values are set/read

# Evolution of programs

- The principle focus on “things that change”
- Separate the data and methods that remain relatively constant throughout the program from those that change
  - isolate parts that change into separate class(es)
  - stable parts are left alone so they are implemented and tested once for all

Principle:

*Don't repeat yourself (DRY)*

# Don't Repeat Yourself Principle (DRY)

- Avoid code duplication
- Common behavior in more places should be abstracted into a class and reused in the common concrete classes
- Satisfy one requirement in one place in the code
- Move common interfaces, data, and operations as high in the inheritance hierarchy as possible



Principle:

*Principle of least knowledge  
(aka Law of Demeter)*

# Principle of Least Knowledge (PLK)

- Also known as law of Demeter
  - “talk only to your immediate friends”
- The complement to strong cohesion in an application is loose coupling
  - classes should collaborate indirectly with as few other classes as possible
- Keep dependencies to a minimum
  - by interacting with only a few other classes, the class is more flexible and less likely to contain errors

## In practical terms...

- A method *foo()* of a class *C* should only call the following types of methods
  - methods of class *C* itself
  - methods of a class passed as an argument
  - method of a class, which is held in instance variable
  - method of any object of the class *C* created locally in method *foo()*
- More importantly, the method *foo()* should not invoke methods on objects that are returned by any subsequent method calls specified above
  - "talk to friends, not to strangers"

Principle:

*Favor composition over  
inheritance*

# Inheritance

- Inheritance for extending functionality and code reuse
  - “is a” relationship
- Inherent disadvantages
  - it breaks encapsulation, since it exposes a subclass to implementation details of its superclass
  - it offers "White-box" reuse, since internal details of super-classes are often visible to subclasses
  - subclasses may have to be changed if the implementation of the superclass changes due to tight coupling
  - implementations inherited from super-classes can not be changed at runtime due to tight coupling

# Object composition

- New functionality and reusability obtained by an object composed of other objects i.e. using object composition
  - “has a” relationship
- Contained objects can be accessed by the containing class only through their interfaces
  - good encapsulation and “black-box” reuse, since internal details of contained objects are not visible
  - fewer implementation dependencies
  - each class is focused on just one task
  - composition can be defined dynamically at run-time through objects using references to other objects of the same type

Principle:

*Program to an interface not  
to an implementation*

# Inheritance revisited

- Class inheritance (implementation inheritance); an object's implementation is defined in terms of another's objects implementation
  - in Java it is the extension from a concrete class
- Interface inheritance (subtyping): describes when one object can be used in place of another object
  - in Java it is the implementation of an interface or the extension from an abstract class



# Interfaces and types

- An interface is the set of methods one object knows it can invoke on another object
  - essentially, it is a subset of all the methods that an object implements
- An interface also defines a specific type of objects
- Different objects can have the same type and the same object can have several different types
- An object is known by other objects only through its interface
  - interfaces allow pluggability

# Interfaces: pros and cons

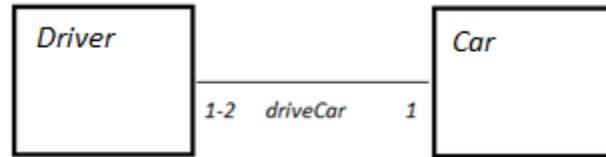
- Advantages
  - clients need not be aware of the specific class of the object they are using
  - an object can be easily replaced by another
  - object connections need not be hardwired to an object of a specific class, thereby increasing flexibility
  - support loose coupling
  - support reuse
  - improves opportunities for composition since contained objects can be of any class that implements a specific interface
- Disadvantages
  - modest increase in design complexity

# Principle:

## *Loose Coupling*

# Relationship: association (*uses a*)

- It defines a strong connection between two classes; e.g. *Driver* has an association with *Car*
  - expressed in UML as a line connecting the classes
  - cardinality specifies how many objects of a class are involved in the relationship



- In Java, association is implemented with data fields and methods

```

public class Driver {
    private Car car;
    public void driveCar(Car c) { .. }
}
  
```

```

public class Car {
    private Driver[] pilots;
    public void setDriver(Driver[] d) { .. }
}
  
```

# Relationship: aggregation (*has a*)

- It is a special form of association representing a relationship of ownership (*has a* relationship) between two classes
- The life-time dependency between the classes is not strong
  - *Car* is still there when the *Driver* dies or sells the *Car*
  - when a container class is destroyed, its contents are not
- This is expressed in UML with a hollow diamond head on the end of the containing class



```
public class ListOfCars {  
    private Car[] cars;  
    ..  
} // aggregating class
```

# Relationship: composition (*owns a*) MCI<sup>®</sup>

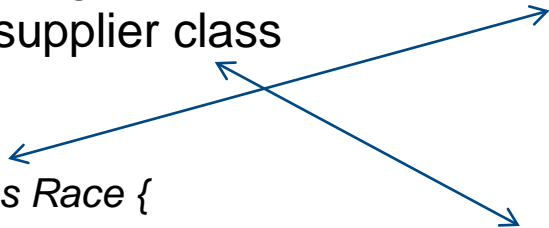
- It is a special form of aggregation and thus represents a strong *has a* relationship between two classes: it is a *owns a*
- The life-time dependency between the classes is strong like e.g. between *Driver* and *Heart* since a *Driver* has a *Heart*
- It is expressed in UML with a filled diamond head on the end of the containing class



```
public class Driver {  
    private Heart heart;  
    private class Heart {  
        // inner aggregated class  
        ...  
    }  
} // aggregating class
```

# Relationship: dependency

- It is a weaker form of relationship which indicates that one class (client) depends on another (supplier) because it uses it at some point of time
- It is implemented using methods in the client class that contains parameters of the supplier class

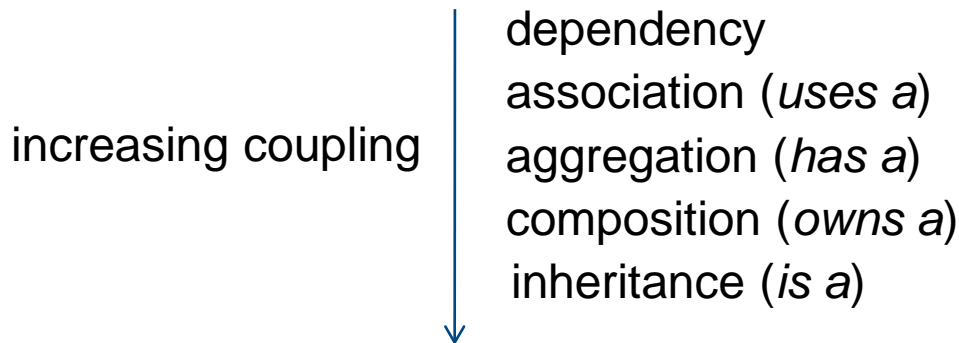


```

public class Race {
    public void setRaceSchedule(Calendar calendar) {
        ..
    }
}
    
```

# Coupling and cohesion

- Coupling is the degree to which one class knows about another class
  - if nothing depends on a class, changes to it has no consequences in the rest of the system i.e. the class is uncoupled from the system
- Cohesion indicates the degree to which a class has a single, well-focused purpose (related to the single responsibility principle)
- Dependency, association, aggregation and composition are binary relationships that differ in the degree of coupling:





# Coupling and cohesion

- Coupling is all about how classes interact with each other while cohesion focuses on how a single class is designed
- Good OO design calls for:
  - loose (not none) coupling so that changes in one location do not to propagate to the rest of the code
  - high cohesion: enforced by methods operating on data members
- All interactions between objects should use the APIs
  - such a hiding of object data and methods by making them accessible only through the methods defined in the class enforces low coupling

# Low coupling & high cohesion

- Mechanisms that promote low coupling:
  - private fields: renaming a field does not influence anything outside the class
  - non-public classes: which can only be used inside a package
- Mechanisms that enable high cohesion:
  - make the members of a class depends on each other
- A good design practice is to:
  - declare instance variables and methods that should not be used outside the class as *private*
  - declare constants, methods to be used outside a class, getters, setters, and most constructors as *public*

# The Principle of Loose Coupling

- Objects that interact should be loosely coupled with well-defined interfaces

## SOLID OOP principles

- Single responsibility principle (SRP)
- Open-closed principle (OCP)
- Liskov substitution principle (LSP)
- Interface segregation principle (ISP)
- Dependency inversion principle (DIP)

SOLID Principle:

*Single responsibility principle*  
(SRP)

# Single responsibility principle (SRP)

- A class should have one, and only one, reason to change
- Each class should have only one responsibility and focus to do one single thing
- SRP promotes
  - code reuse
  - clarity
  - readability
- High cohesion
- <https://dzone.com/articles/single-responsibility>



# Does this break the SRP?

```
public class Student {  
    private String studentId;  
        name,  
        address;  
    public boolean isAdmittedToFinalThesis() {  
        ....  
    }  
    public double calculateStudentGrant() {  
        ....  
    }  
    // Constructors, getters & setters for all the private attributes  
}
```

# Refactoring class to comply with SRP

```
public class SchoolAdmin {
    public boolean isAdmittedToFinalThesis(Student s) {
        ....
    }
}
```

```
public class TaxOffice {
    public double calculateStudentGrant(Student s) {
        ....
    }
}
```

```
public class Student {
    private String studentId;
        name,
        address;
    // Constructors, getters & setters for all the private attributes
}
```



# SOLID Principle:

## *Open-closed principle (OCP)*

# Open-closed principle (OCP)

- Software entities such as methods, classes, and modules should be open for extension but closed for modifications
- Anytime the code is changed, there is a potential for breaking it
  - sometimes the code cannot be changed



# Definitions

- A class is closed if its runtime or compiled class is available for use as a base class which can be extended by child classes
  - changes are guaranteed to not happen
  - the class is open for extension
- A class is open if its functionality can be enhanced by sub-classing it
  - using the Liskov Substitution principle the class can be replaced by one of its sub-classes
  - this sub-class behaves as its parent class but is an enhanced version of it; the said class is open for modification via extension

# Is this closed for modification and open for extension?



```
public class Rectangle {  
    public double length,  
        width;  
}
```

```
public class Circle {  
    public double radius;  
}
```

```
public class AreaCalculator {  
    public double calcRectArea(Rectangle rectangle) {  
        return rectangle.length *rectangle.width;  
    }  
    public double calcCircleArea(Circle circle) {  
        return Math.PI*circle.radius*circle.radius;  
    }  
}
```

# Refactoring

```
public interface Shape {
    public double calcArea();
}
```

```
public class Rectangle implements Shape {
    double length,
        width;
    public double calcArea(){
        return length * width;
    }
}
```

```
public class Circle implements Shape {
    public double radius;
    public double calcArea(){
        return Math.PI*radius*radius;
    }
}
```

```
public class AreaCalculator {
    public double calcArea(Shape shape) {
        return shape.calcArea();
    }
}
```

SOLID Principle:

*Liskov substitution principle*  
(LSP)

# Liskov substitution principle (LSP)

- Objects should be replaceable with instances of their subtypes without altering the correctness of that program
- It leads to design by contract
- Java inheritance mechanism follows Liskov Substitution Principle because a superclass reference can hold a subclass object
- Program to an interface not to an implementation!



SOLID Principle:  
*Interface segregation  
principle (ISP)*



# Interface segregation principle (ISP)

- Many client-specific interfaces are better than one general-purpose interface
- “fat” classes/interfaces should be split into smaller and more specific ones
- The goal is to keep a system decoupled i.e. easier to
  - modify
  - refactor
  - redeploy



## Does it break ISP?

```
public interface IntfPrinter {  
    void print();  
    void fax();  
    void scan();  
}
```

```
public class AModernPrinter implements IntfPrinter {  
    void print() { ... }  
    void fax() { ... }  
    void scan() { ... }  
}
```

## Does it break ISP?

```
public class AnOldPrinter implements IntfPrinter {
    void print() { ... }
    void fax() { ... }
    void scan() {
        throw new NotSupportedException();
    }
}
```

# Refactoring

```
public interface IntfPrinter {
    void print();
}
public interface IntfFax {
    void fax();
}
public interface IntfScanner {
    void scan();
}
```

```
public class AnOldPrinter
    implements IntfPrinter, IntfFax {
    void print() { ... }
    void fax() { ... }
}
```

SOLID Principle:

*Dependency inversion  
principle (DIP)*

# Dependency inversion principle (DIP)

- Depend upon abstractions and do not depend upon concretions
- Higher level modules should not depend upon low level modules
  - both should depend on abstractions
- Abstractions should not depend on details while instead details should depend on abstractions



## Example

```
public class Logger {  
    private FileSystem fileSystem = new FileSystem();  
  
    public void log(String txt) {  
        fileStream = fileSystem.openFile("log.txt");  
        fileStream.write(txt);  
        fileStream.close();  
    }  
}
```

# Refactoring

```
public interface IntfLoggable {
    void log(String txt);
}

public class FileSystem implements
    IntfLoggable {
    public void log (String txt) {
        ...
    }
}
```

```
public class Logger {
    private IntfLoggable logger;
    public Logger (IntfLoggable logger) {
        this.logger = logger;
    }
    public void log(String txt) {
        logger.log(txt);
    }
}
```



## More specific guidelines

- Do not add public members that are inconsistent with the interface abstraction
- Avoid putting private implementation details into the class's interface
- Only inherit if the derived class is a more specific version of the base class
- Be suspicious of classes of which there is only one instance
- Be suspicious of base classes that only have a single derived class
- Eliminate data-only classes
- Eliminate operation-only classes