HBO Graduaat Informatica Optie Programmeren

Java Basics

Java and OOP

cvo leerstad

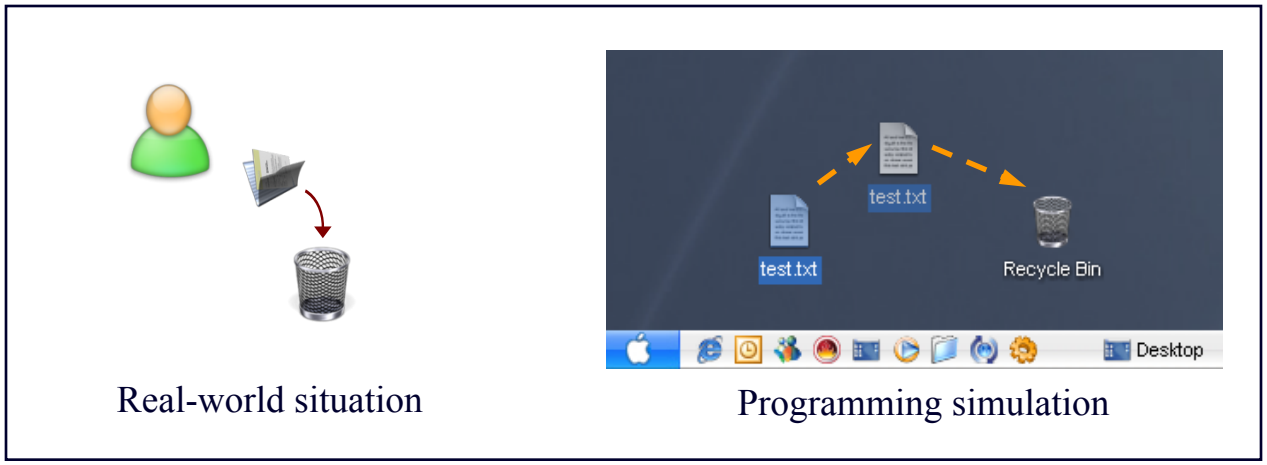
volwassenenonderwijs

Programming by simulation

The purpose of Object-Oriented programming
is to simulate the objects
of the real world
by software objects

Programming by simulation

- A simple example:
  - Look for a file
  - Put the file into the dustbin
  - Eventually recover the file



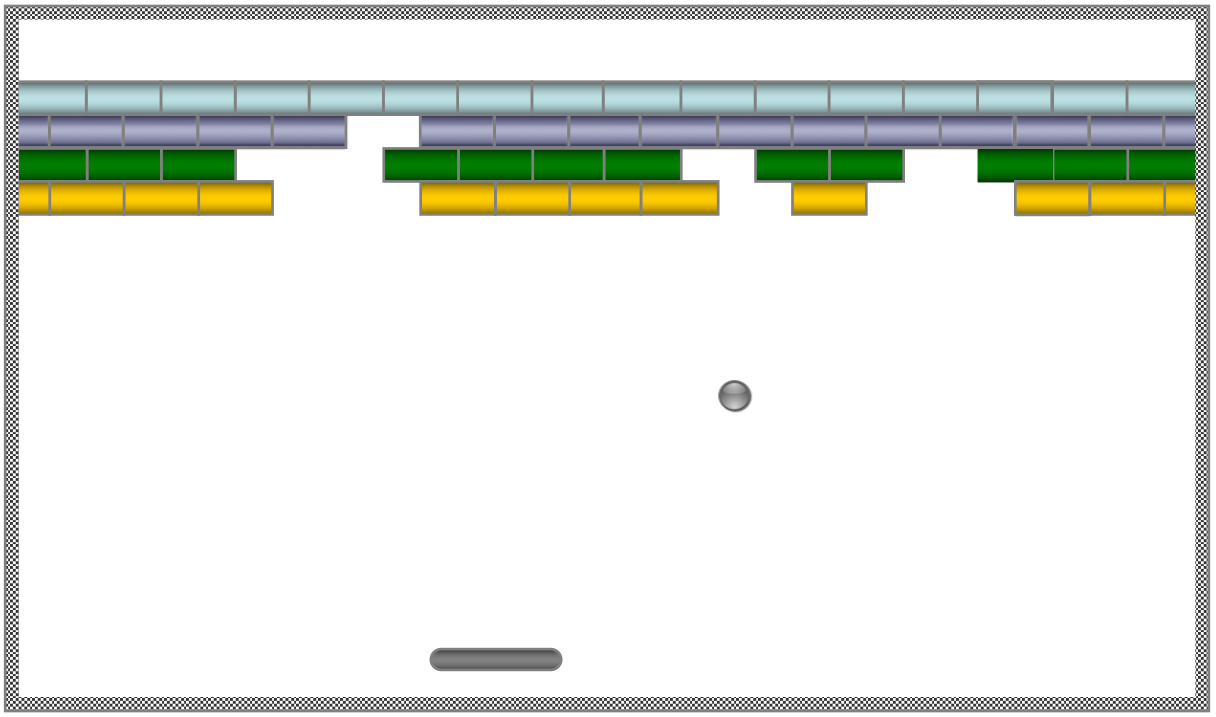
Real-world situation

Programming simulation

The operations on the desktop objects mimic the way their real-world counterparts are manipulated

Programming by simulation

- The video game example:
  - How to identify the software objects?
  - How these objects interact together?



Programming by simulation

- Identify the video game software objects:
  - The wall
  - The bricks
  - The paddle
  - The ball

Programming by simulation

- The video game itself

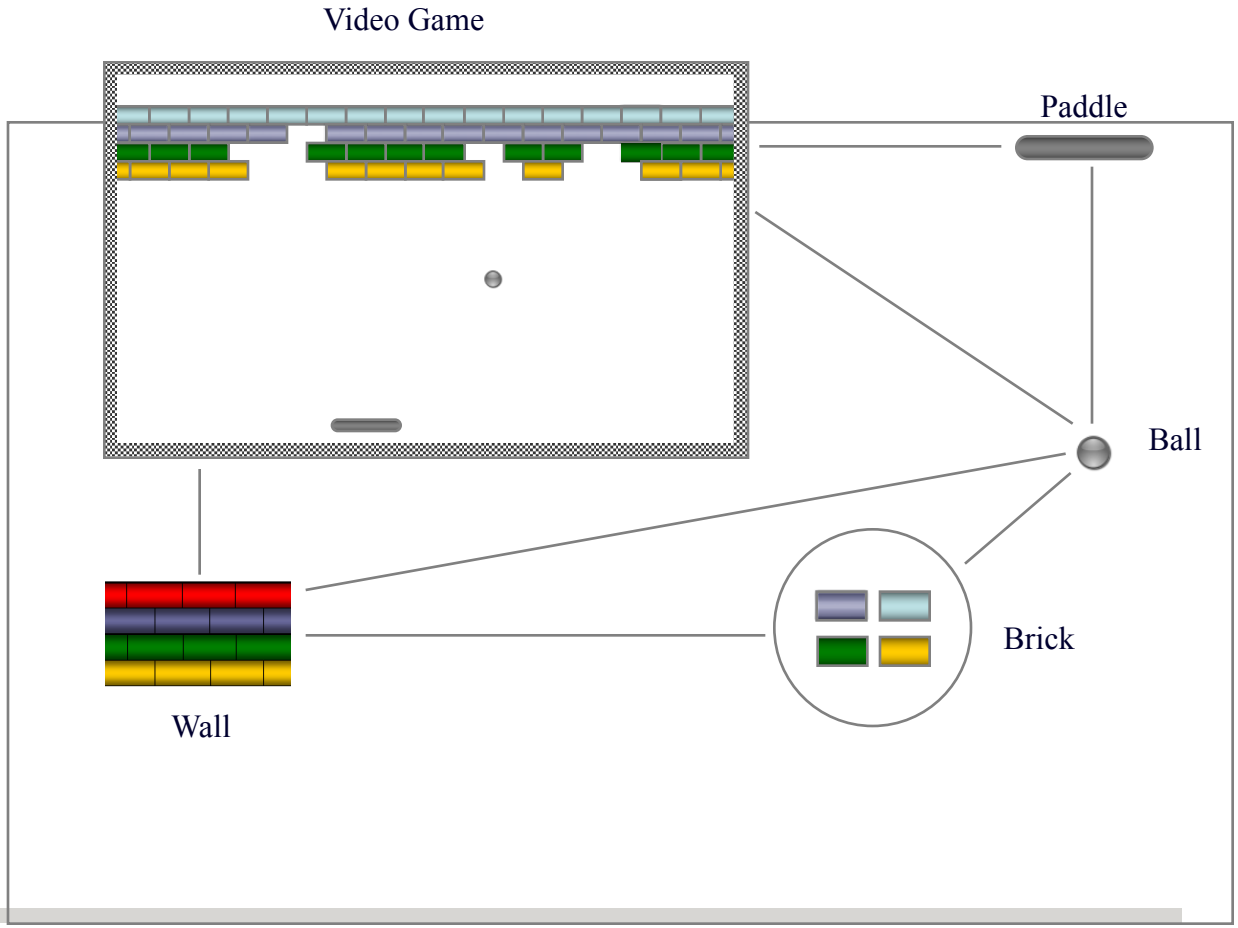
A real world object is a good candidate to be a software object

- Identify the objects relationships:
  - The ball and the paddle
  - The ball and the wall …

To discover the interactions, simulate the video game

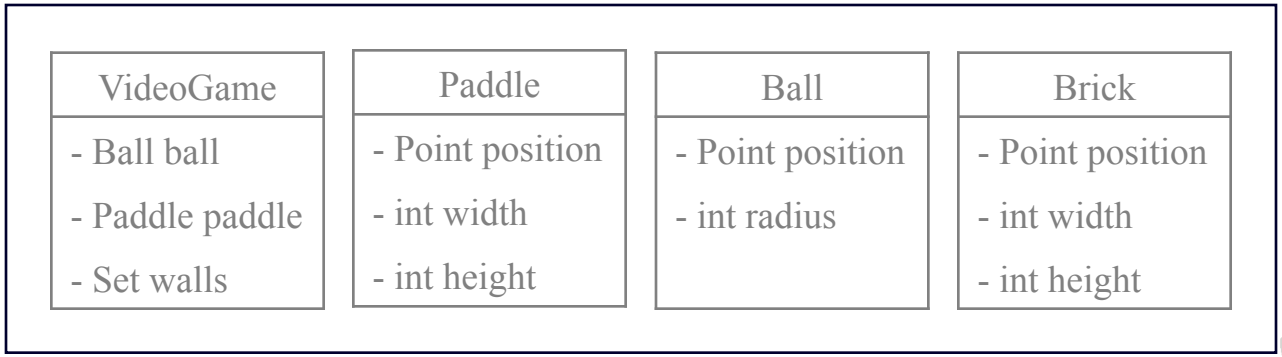
Programming by simulation

- The video game software objects and relationships



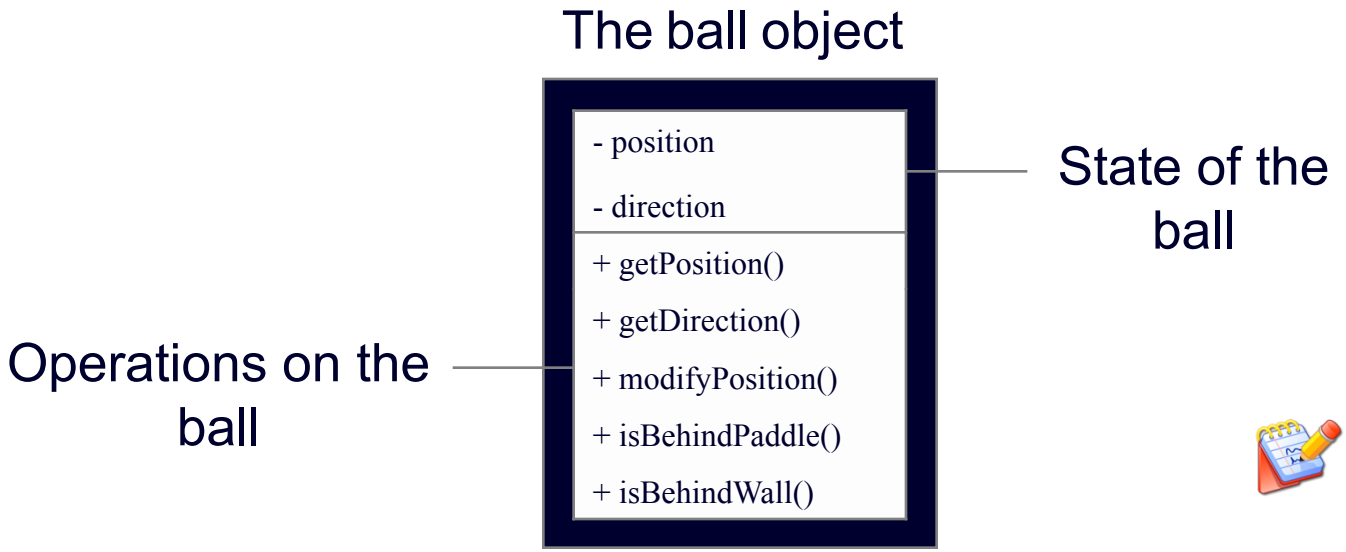
Programming by simulation

- Each object is characterized by its state
  - A ball might consist of a radius and a position
  - A paddle might be characterized by a height, a width, and by its position
  - …



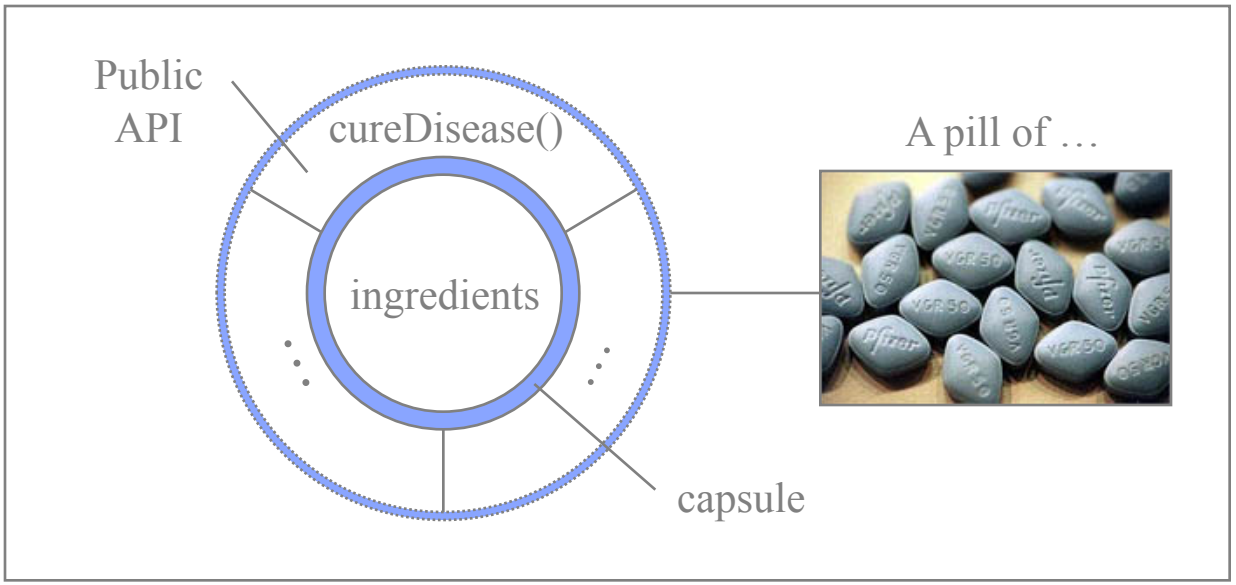
Programming by simulation

- Each object supports a set of operations that can be performed on its state



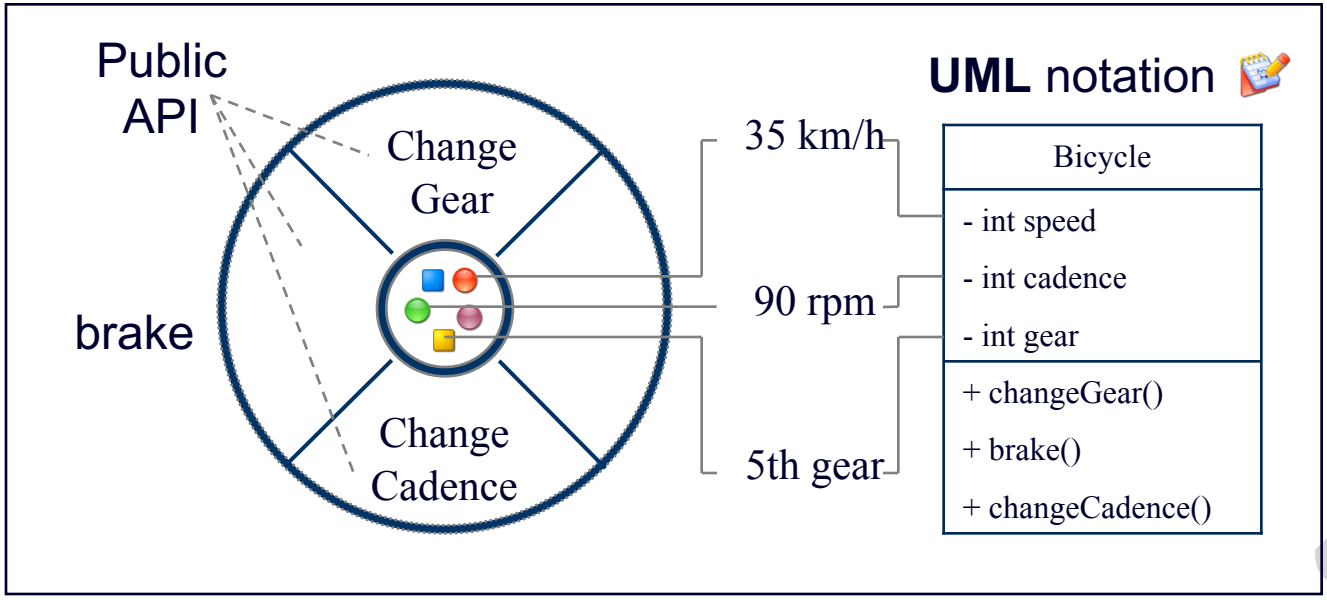
The encapsulation principle

- Definition
  - The ability to group together in an object :
    - The state of the object
    - The allowable operations on that state



The encapsulation principle

- Another example:
  - The conceptual view of a bicycle



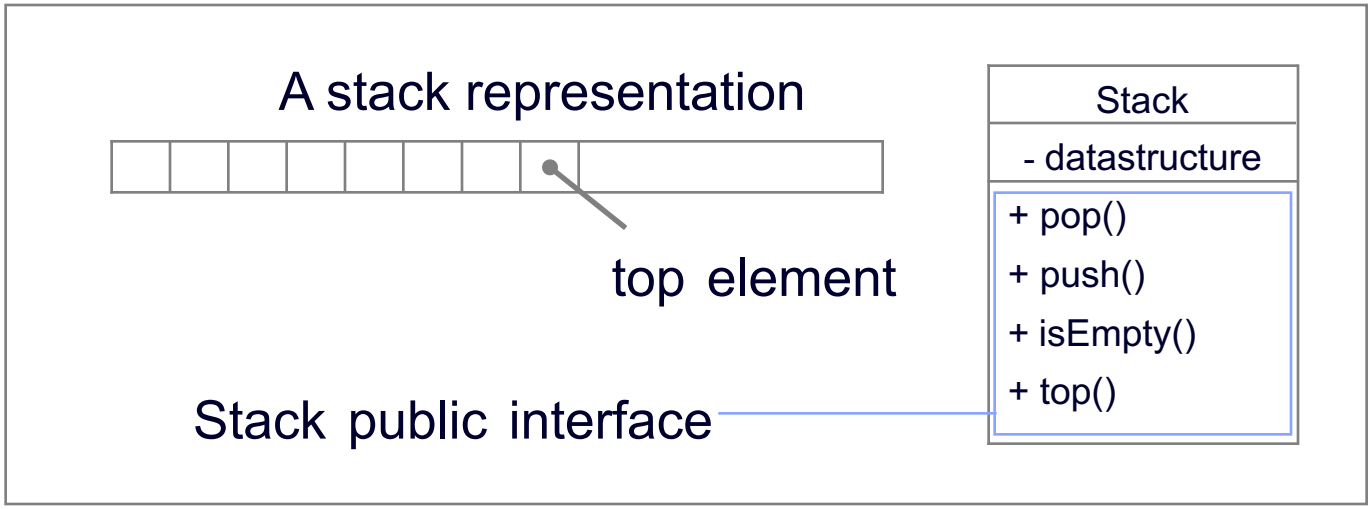
The encapsulation principle

- Information hiding:
  - To access or modify the state of an object, other objects have to use its public interface.
  - The public API implementation can change without affecting the other objects that depend on it.

- Modularity
  - Separation of the object's user interface from its implementation lead to maintainable and reusable software.
  - Implementers can modify the implementation of an object, in a manner that is transparent to users.

The encapsulation principle

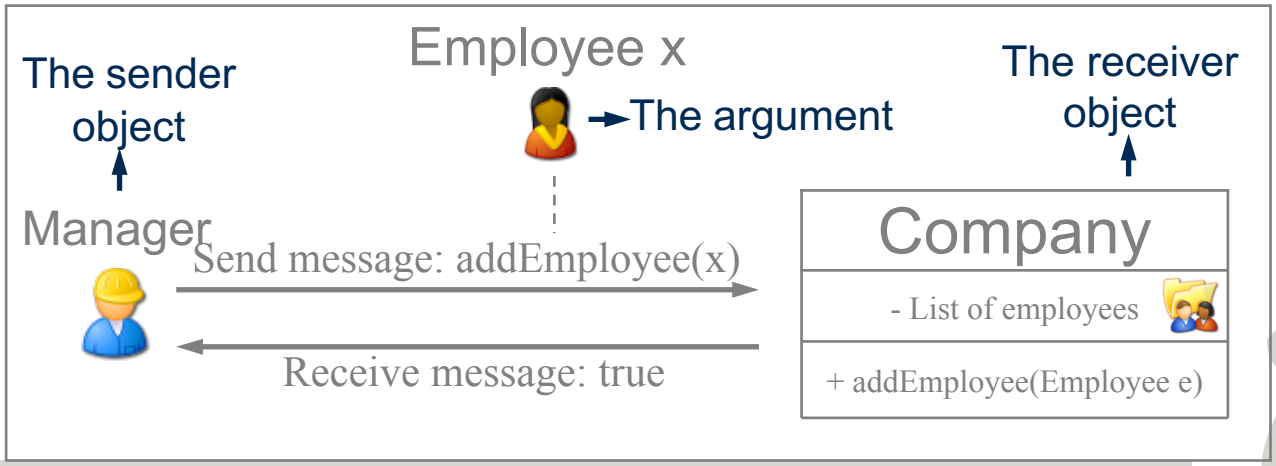
- Abstraction of the problem domain:
  - You can use objects without knowing the detail of their internal representation



No matter the data structure used by the stack implementor, **users always invoke the same public interface**

The encapsulation principle

- Message-passing communication
  - Objects communicate together by message-passing:
    - The sender object call the public API, or ***methods***, of the receiver object and eventually wait for reply
    - The sender object can also pass ***arguments*** to the receiver, to fulfill the request



The encapsulation principle

- Definitions
  - Method
    - Operation invoked when a message is received by an object
    - Its definition describes how the object will react upon receiving the message
  - Protocol
    - The set of messages to which an object responds

Message-passing is synchronous. A second message cannot be sent after the result of sending a first message has been returned

The encapsulation principle

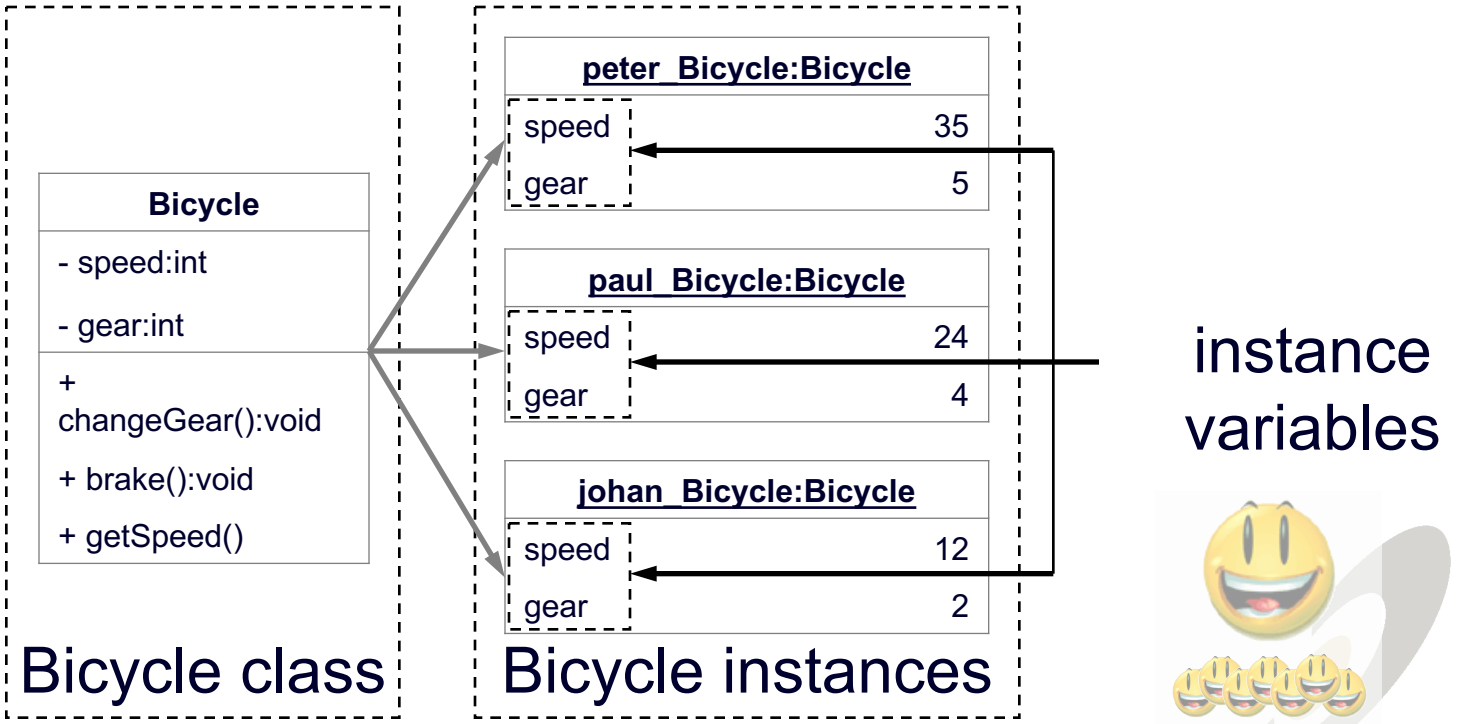
- Definitions
  - Class
    - Representation of particular object type, including its state and its public API
    - Act as a ***mould*** from which you can produce ***individual objects***, named ***instances***, with similar characteristics, attributes, and behaviors
    - Repository for methods that can be executed by all instances belonging to that class

The encapsulation principle

- Definitions
  - Instances
    - ***Individual*** object described by a particular class, and considered as ***member*** of that class
    - Repository for data that describes the state of an individual object
    - Can contain ***instance variables***, encapsulated inside the object, and accessible directly by the object itself, or from other objects through the object's public API

UML Notation

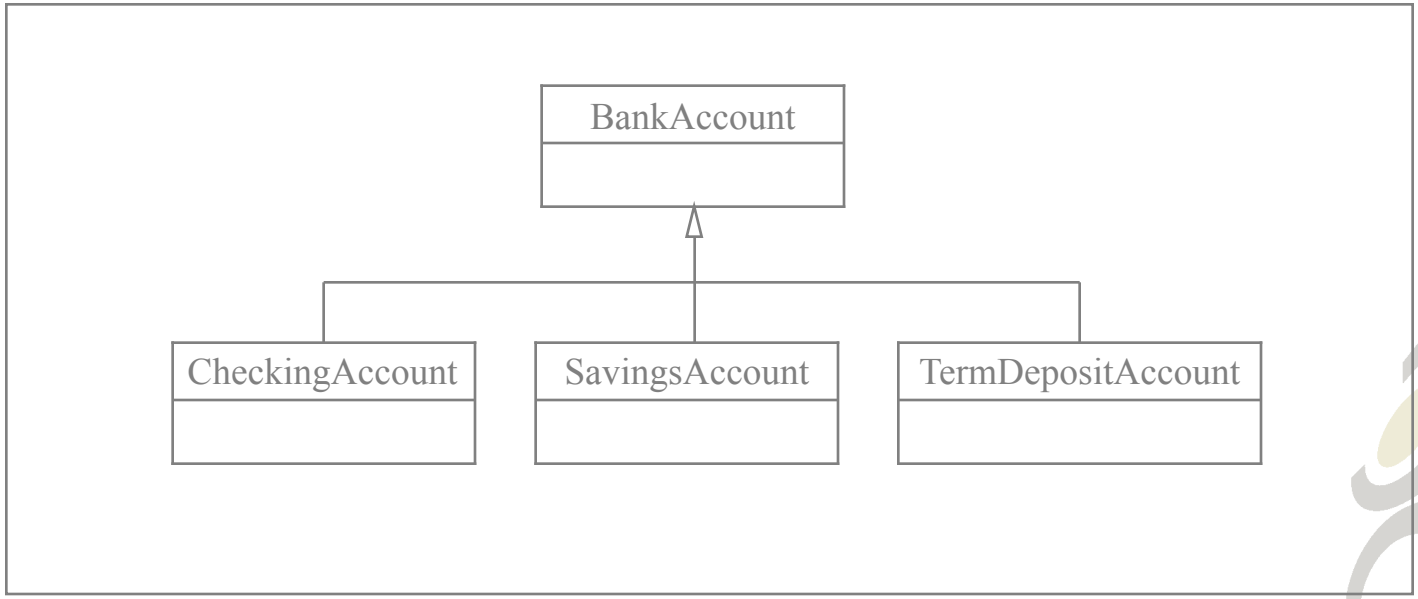
- UML representation for classes and named instances



The inheritance principle

- Definition
  - Transfer of characteristics of a class in object oriented programming to the classes derived from it

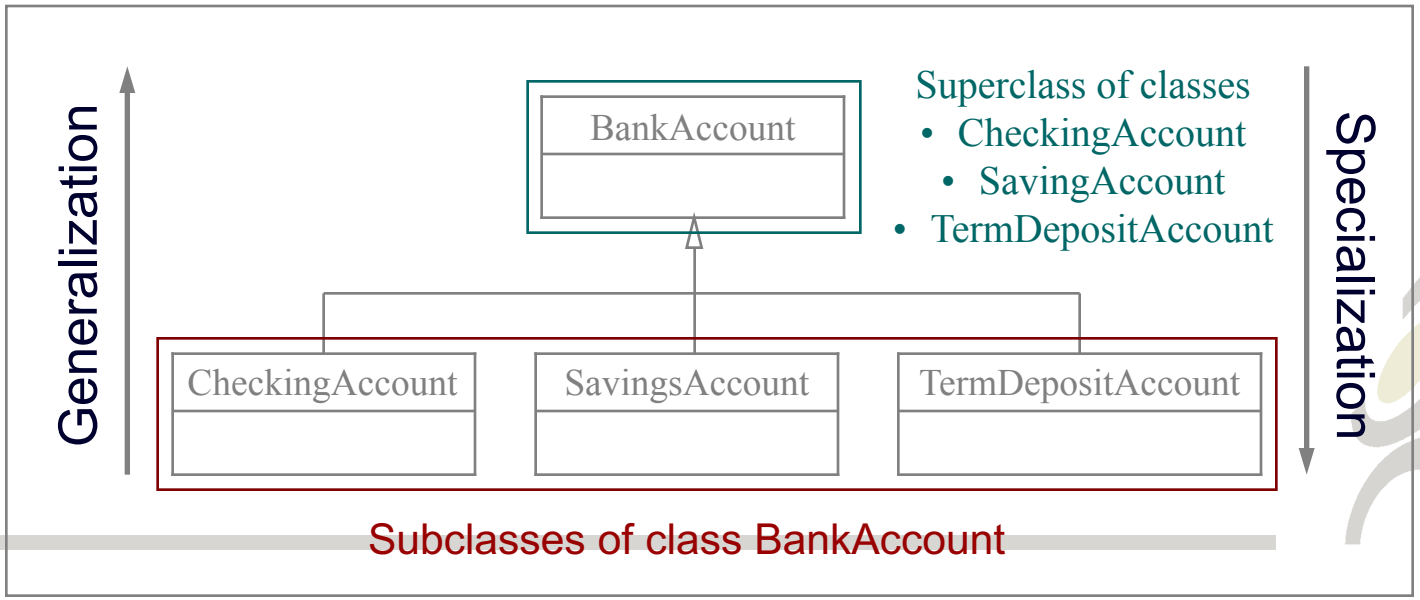
A bank accounts hierarchy



The inheritance principle

- Definition
  - Transfer of characteristics of a class in object oriented programming to the classes derived from it

A bank accounts hierarchy



The inheritance principle

- Inheritance is unilateral and could be translated into « ***is a*** » relationship:
  - A CheckingAccount « ***is a*** » BankAccount
  - A SavingsAccount « ***is a*** » BankAccount
  - A TermDepositAccount « ***is a*** » BankAccount

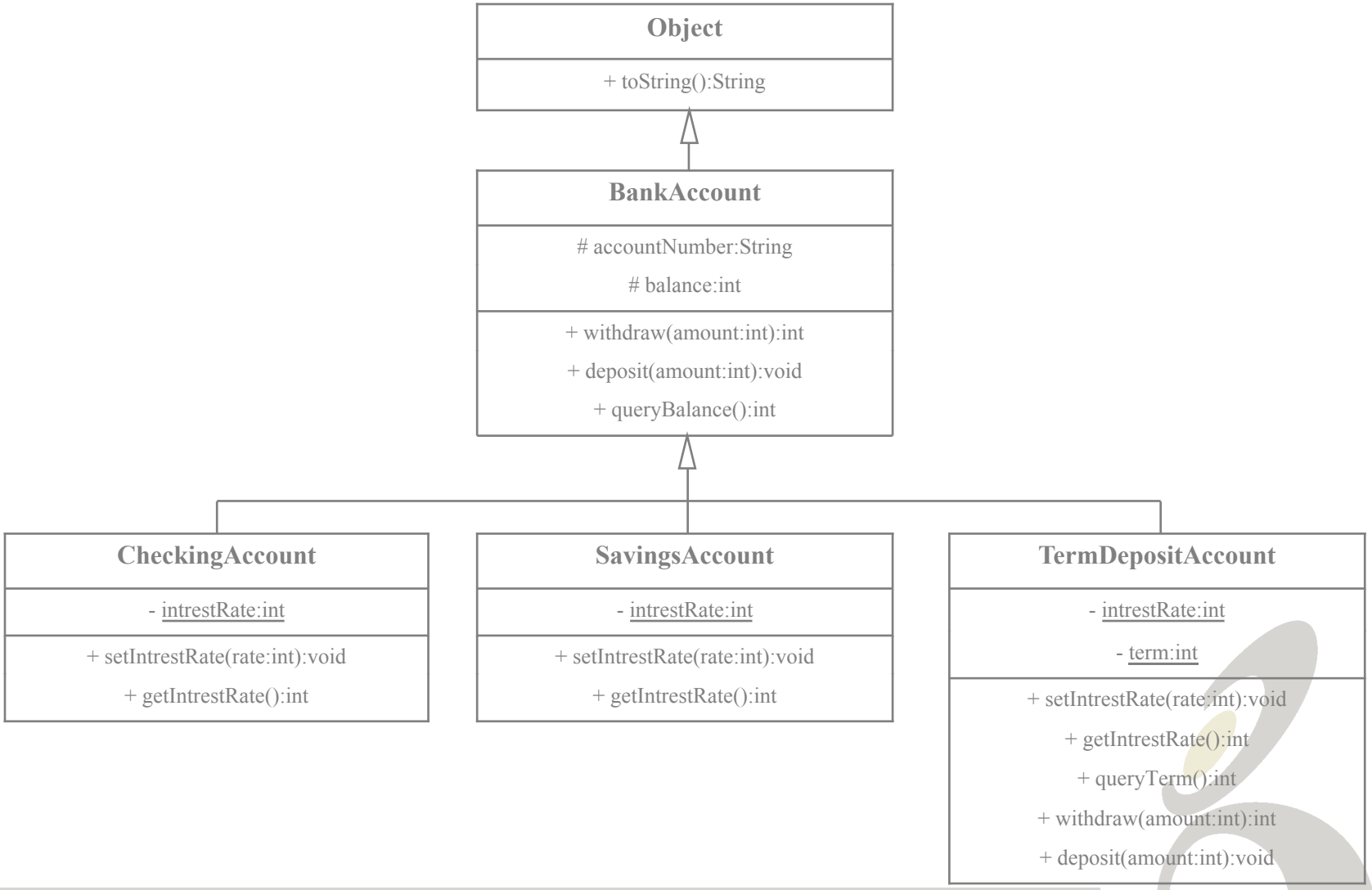
All these Accounts types inherit from the BankAccount variables and methods:

- An account number, a balance,…
- A method to query the balance, to deposit money,…

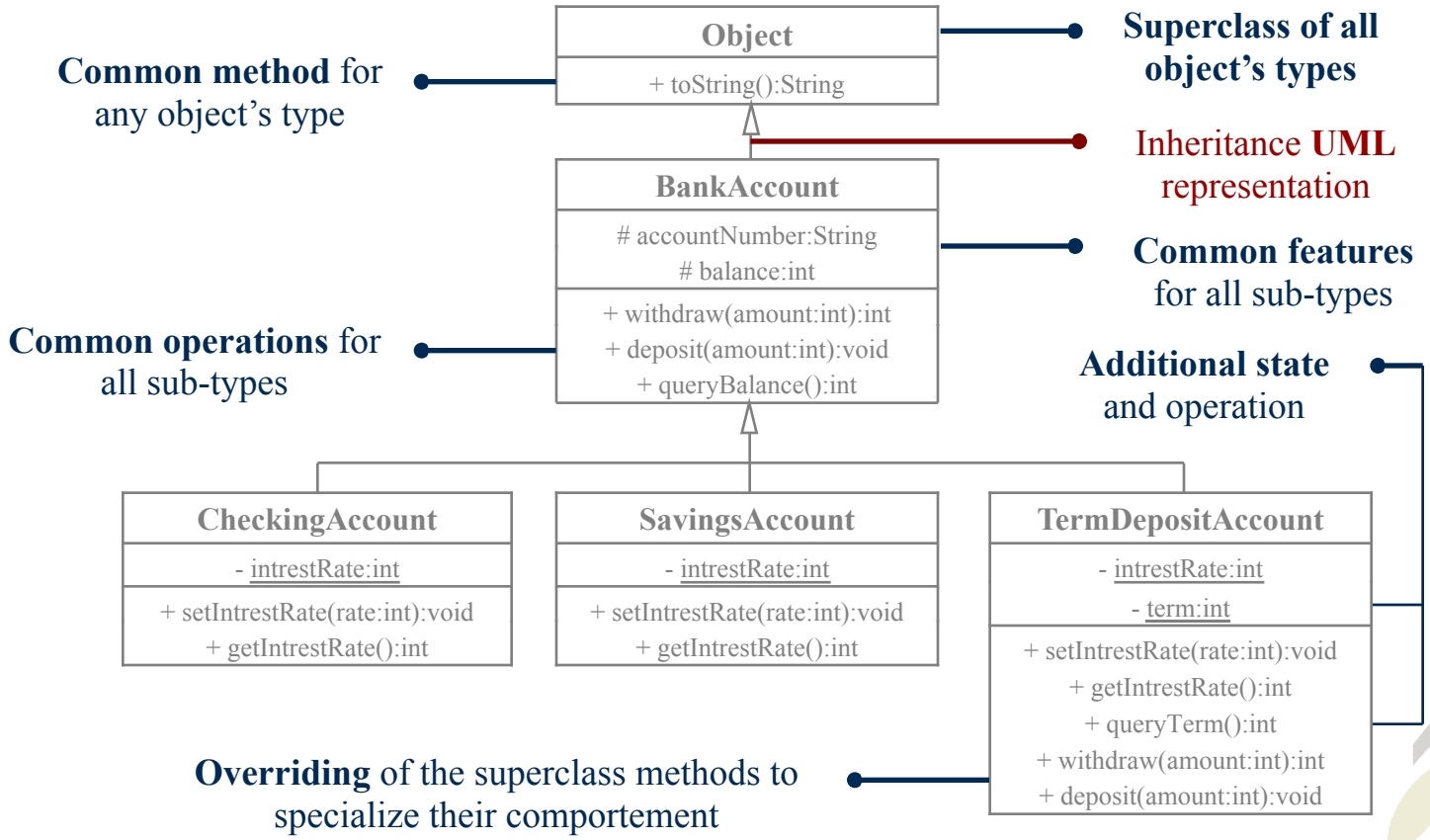
The inheritance principle

- Definitions
  - Subclass
    - A class that inherits methods and representation from an existing object class
    - Specialize an object class with additional variables or methods
  - Superclass
    - A class from which another object class inherits representation and methods
    - Intended to maintain all common features of its subclasses

The inheritance principle



The inheritance principle



The inheritance principle

- What about message passing?

Object receiver	Message	Executed operation
TermDepositAccount	queryTerm()	*queryTerm()* in class **TermDepositAccount**
CheckingAccount	queryBalance()	*queryBalance()* inherited from **BankAccount**
SavingsAccount	queryTerm()	**ERROR** no method in the class or in the superclasses
SavingsAccount	toString()	*toString()* inherited from **Object** class
TermDepositAccount	withdraw()	*withdraw()* **specialized** in class **TermDepositAccount**

The inheritance principle

- When create a subclass?
  - Support of ***additional operations***, other than those inherited from the superclass
  - Necessity to ***override existing operations*** supported by the superclass, which are inappropriate for the new object type
  - Necessity for the subclass to ***maintain additional state***

Inheritance

- Definition
  - A subclass inherit of the data and methods of the superclass(es).
  - The programmer can use the inherited methods and data without redefine them.
  - It’s not only for the speed of development but also ensure an inherent validity

Inheritance

- How to use it in Java ?

```
public class A {
 int i = 10;
 public int add(int value) {
  return i + value;
 }
}
public class B extends A {}

public class Demo {
 public static void main(String[] args) {
 B b = new B();
 int valToAdd = 78;
 int result = b.add(valToAdd);
 System.out.println("the result is "+result);
 }
}
```

The result is 88

Assignment compatibility and type conversion

- Assignment compatibility
- Type conversion and cast operator
  - automatic or cast operator
  - Cast Operator
    - int i = 3;
    - float f = (float)i;
- Applying a cast operator
  - doesn’t change the type of a variable

Assignment compatibility and type conversion

- Reminder : Primitive values and type conversion
  - boolean value can only be assigned to boolean type
  - Primitive value can be assigned to any variable whose range is wider ⇔ can't be assigned to a variable whose range is narrower (or with cast operator).

Assignment compatibility and type conversion

- Assignment compatibility for references
  - an object instantiated can be assigned to :
    - the same class
    - the superclass
    - the interface implemented
    - the interface whose a superclass implemented
  - an assignment doesn't required a cast operator
  - Type Object is completely generic
    - the Object class is a superclass of every other class.

Assignment compatibility and type conversion

- Assignment compatibility for references
  - an object instantiated can be assigned to :
    - the same class
    - the superclass
    - the interface implemented
    - the interface whose a superclass implemented
  - an assignment doesn't required a cast operator

```
B b = new B();
B c = b // this will compile an run without error
A a = b; // this will compile if class B is a subclass
of A;
Object obj = b; // this will allways work because Object
is ...
```

Assignment compatibility and type conversion

- Converting reference types with a cast
  - Other assignments require cast operator
  - The cast can only be performed in a same class or interface hierarchy.
    - Object class downcast to a other class.
  - If your cast is not correct you get an exception at run time.
  - Using of “instance of”

```
Object obj = new B();
if ( obj instanceof  B) {
 B b = (B) obj;
}
```

Polymorphism

- Definition
  - Polymorphism means the possibility to treat an object of any subclass of a base class, as if it were an object of the base class
  - How ?
    - Method overloading
    - Method overriding through inheritance
    - Method overriding through interface

Polymorphism

- Overloading
  - the same name of method has different effects
  - Overloading a method in different classes which are not related
  - Overloading a method in a same class
    - the methods have the same name but different parameters list
    - the compiler distinguish them by their parameters list
    - name + parameters list + parameters order = ***signature***

Polymorphism

- Overloading
  - the same name of method has different effects
  - Overloading a method in different classes which are not related
  - Overloading a method in a same class
    - the methods have the same name but different parameters list
    - the compiler distinguish them by their parameters list
    - name + parameters list + parameters ***signature***

DataRenderer
+ draw(int i)
+ draw(String s)
+ draw(float f)

Polymorphism

- Overriding
  - Resolved dynamically at runtime
  - Same signature between superclass method and overriding method
  - Overriding to be more public

Polymorphism

- Overriding : Example without Polymorphism

```
public class Shape {
 private static final int RECT = 1;
 private static final int SQUARE = 2;
 private static final int CIRCLE = 3;
 private int type;

 public Shape(int aType) {
 type = aType;
 }
 public String draw() {
   switch (type) {
   case CIRCLE:
   return "I am a circle.";
   case SQUARE:
   return "I am a square.";
   case RECT:
   return "I am a rectangle.";
   default:
   return "Undefined Object.";
   }
 }

public static void main (String[] args){
 Shape firstShape = new Shape(CIRCLE);
 Shape secondShape = new Shape(RECT);
 Shape thirdShape = new Shape(SQUARE);
 System.out.println(firstShape.draw());
 System.out.println(secondShape.draw());
 System.out.println(thirdShape.draw());
 }
}
```

Polymorphism

- Overriding : Example without Polymorphism

```
public class Shape {
 private static final int RECT = 1;
 private static final int SQUARE = 2;
 private static final int CIRCLE = 3;
 private int type;

 public Shape(int aType) {
 type = aType;
 }
```

```
 public String draw() {
   switch (type) {
   case CIRCLE:
   return "I am a circle.";
   case SQUARE:
   return "I am a square.";
   case RECT:
   return "I am a rectangle.";
   default:
   return "Undefined Object.";
    }
 }

public static void main (String[] args){
 Shape firstShape = new Shape(CIRCLE);
 Shape secondShape = new Shape(RECT);
 Shape thirdShape = new Shape(SQUARE);
 System.out.println(firstShape.draw());
 System.out.println(secondShape.draw());
 System.out.println(thirdShape.draw());
 }
}
```

Polymorphism

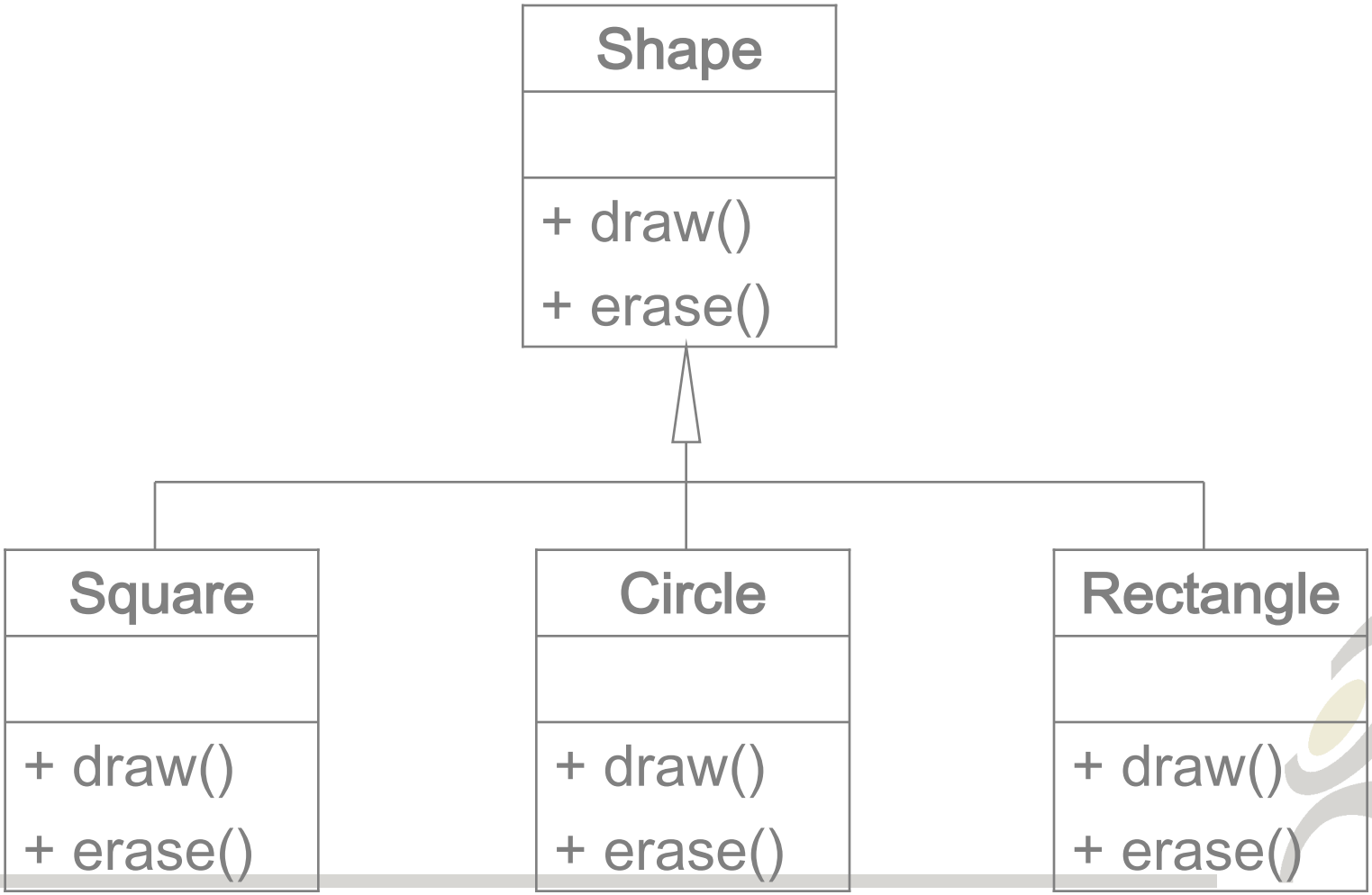
- Overriding : Example with Polymorphism

```
public class Shape {
public String draw(){
    return "Undefined Object.";
}}

public class Rectangle extends Shape {
public String draw(){
    return "I'm a Rectangle";
}}

public class Circle extends Shape {
public String draw(){
    return "I'm a Circle";
}}

public class Square extends Shape {
public String draw(){
    return "I'm a Square";
}}
```

```
public class Test {
 public static void main (String[] args){
    Shape firstShape = new Rectangle();
    Shape secondShape = new Circle();
    Shape thirdShape = new Square();

System.out.println(firstShape.draw());

System.out.println(secondShape.draw());

System.out.println(thirdShape.draw());
  }
}
```

Polymorphism

- Overriding : Example with Polymorphism

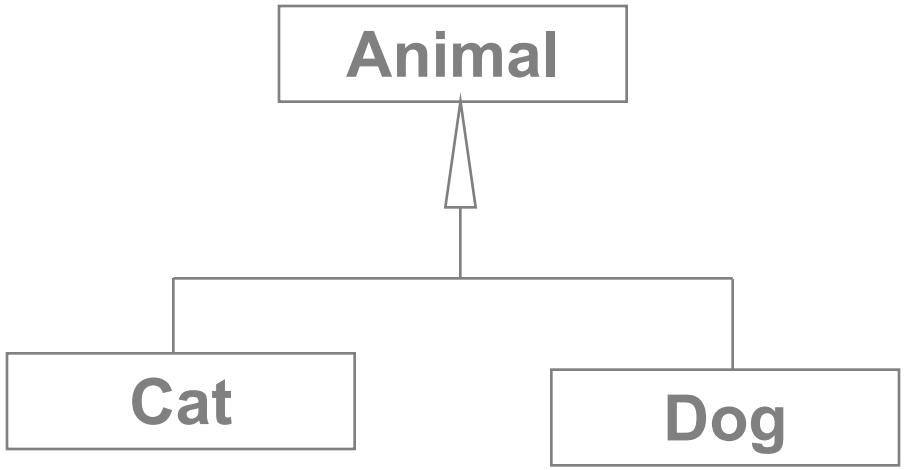


Polymorphism

- The difference between
  - Overloading
  - Overriding

Dynamic binding

- Introduction



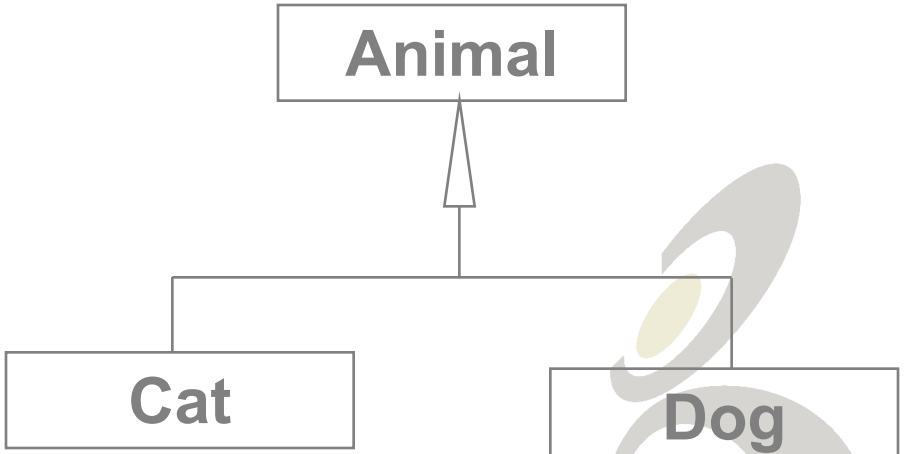
Dynamic binding

- Polymorphism and dynamic binding
  - If you want an Animal:

```
Animal a = new Animal();
```

  - With polymorfism :

```
Animal animal = new Dog();
Animal animal = new Cat();
```



Dynamic binding

- Polymorphism and dynamic binding
  - How obtain full benefit of polymorphism
    - Send a message without knowing object's class
    - In Java, invoke a method defined in the superclass
      - The object must be of superclass or subclass
      - If the object is of subclass, there is a ***run-time coupling***. ***Dynamic binding***
    - The JVM invokes the method in the real object class if present.

The class Object

- Introduction
  - Class Object is superclass of all classes
  - You may override
    - **toString();**
    - **equals(Object o)**
    - **hashCode ()**
    - finalize()
    - clone()
  - You can’t override (final)
    - getClass()
    - notify()
    - notifyAll()
    - wait()

The class Object

- The toString() method
  - Object's toString() returns a String representation of the object.
    He
    ha
  - If y
    mu

```
public class Buyer {

  private String buyerNumber;
  private String name;

  public Buyer(String buyerNumber, String name){
   this.buyerNumber = buyerNumber;
   this.name = name;
  }

  public String toString() {
   return this.buyerNumber + " - " + this.name;
  }
}
```

The class Object

- The equals(Object o) method
  - 2 instances = ?

The class Object

- The equals(Object o) method
  - 2 instances = ?

```
public class Example {
  public static void main(String[] args) {
    Buyer p1 = new Buyer("07/12345", "Vanzwan");
    Buyer p2 = new Buyer("07/12345", "Vanzwan");
    System.out.println("these 2 persons are equals : " + (p1 == p2));
  }
                                                    }
```

these 2 persons are equals : false
Why ?

Because reference (memory address) is not equals

The class Object

- The equals(Object o) method
  - 2 instances = ?

```
public class Example {
  public static void main(String[] args) {
    Buyer p1 = new Buyer("07/12345", "Vanzwan");
    Buyer p2 = new Buyer("07/12345", "Vanzwan");
    System.out.println("these 2 persons are equals : "+(p1.equals(p2)));
  }
}
```

this 2 persons are equals : false

See equals of Object class

```
public boolean equals(Object o){
return (this == o);
}
```

The class Object

- The equals(Object o) method
  - 2 instances = ? (use of equals(Object o))

```
public class Buyer {
{
        ...
public boolean equals(Object o) {
  if ((null == o) && (!(o instanceof Buyer)))
return false;
Buyer temp = (Buyer) o;
return
  ((this.buyerNumber.equals(temp.getBuyerNumber()) &&
  (this.name.equals(temp.getName()))));
}
}
```

this 2 persons are equals : true

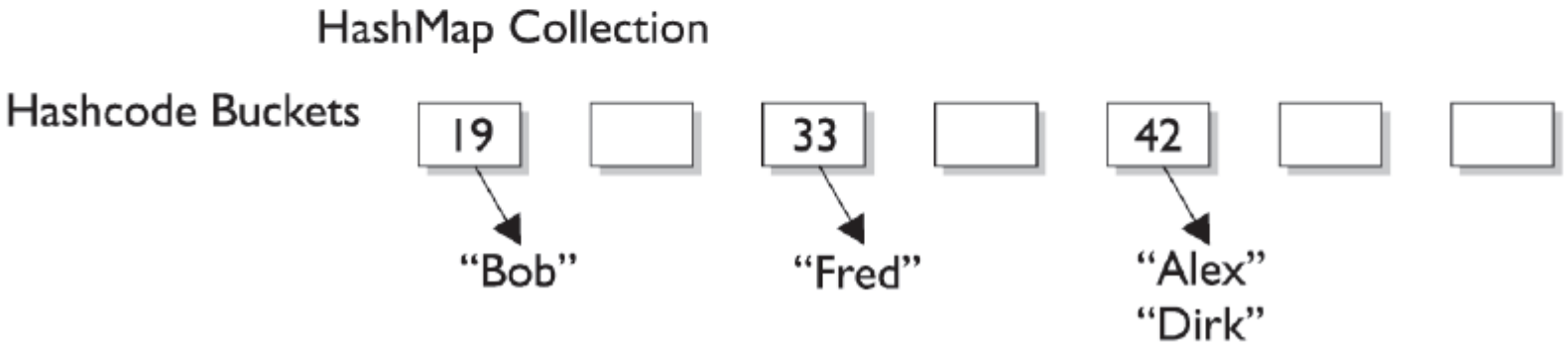
The class Object

- The hashCode method
  - Hashcode value is used by some collection classes (later more on that)
  - It isn't necessarily unique

The Object Class

- The hashCode method

Key	Hashcode Algorithm	Hashcode
Alex	A(1) + L(12) + E(5) + X(24)	= 42
Bob	B(2) + O(15) + B(2)	= 19
Dirk	D(4) +I(9) + R(18) + K(11)	= 42
Fred	F(6) + R(18) + E(5) + (D)	= 33



The Object Class

- The `hashCode()` contract
  - Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode()` method must consistently return the same integer, provided no information used `in equals()` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

The Object Class

- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode()` method on each of the two objects must produce the same integer result.
- It is NOT required that if two objects are unequal according to the `equals`(java.lang.Object) method, then calling the `hashCode()` method on each of the two objects must produce distinct integer results.

The Object Class

- The hashCode() contract

Condition	Required	Not Required (But Allowed)
x.equals(y) == true	x.hashCode() == y.hashCode()	
x.hashCode() == y.hashCode()		x.equals(y) == true
x.equals(y) == false		No hashCode() requirements
x.hashCode() != y.hashCode()	x.equals(y) == false	

Conclusion

- Object-Oriented thinking implies:
  - To simulate the real-world objects using software objects
  - To ensure interactions between software objects by message-passing
  - To be aware of the three fundamental principles:
    - ***Encapsulation***
    - ***Inheritance***
    - ***Polymorphism***

Questions ??