Introduction to Object Oriented Programming

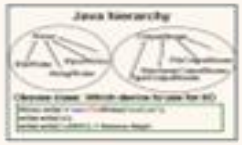
Roy Schwartz, The Hebrew University (67125)

Lecture 13:

Summary

(Partial) Summary

Intro to java
Class vs. object
Encapsulation
Inheritance
Polymorphism
Abstract class vs. interface
Design Patterns
Collections
Generics
Packages
Exceptions
Nested Classes
Modularity
Streams
RegExps
Serialization
Reflection
...

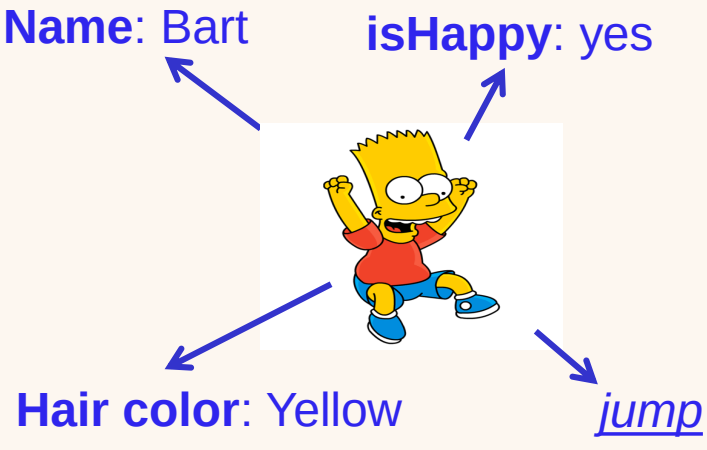


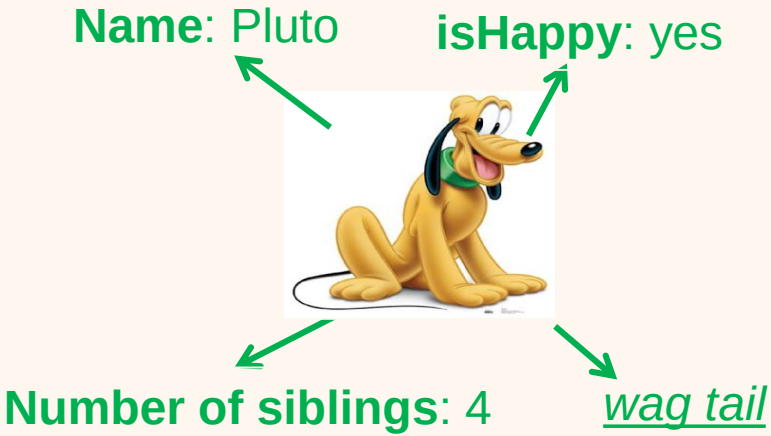
Lecture 13a: Overview

- Part a: **Introduction to java** (Lectures 1-2)
- Part b: **Polymorphism and Basic Design** (Lectures 3-5)
- Part c: **Core Topics in java** (Lectures 6-7)
- Part d: **Modularity and Advanced Design** (Lectures 8-9)
- Part e: **Advanced Topics** (Lectures 10-13)

What is an Object?

- Real-world objects share two characteristics: They all have *state* and *behavior*



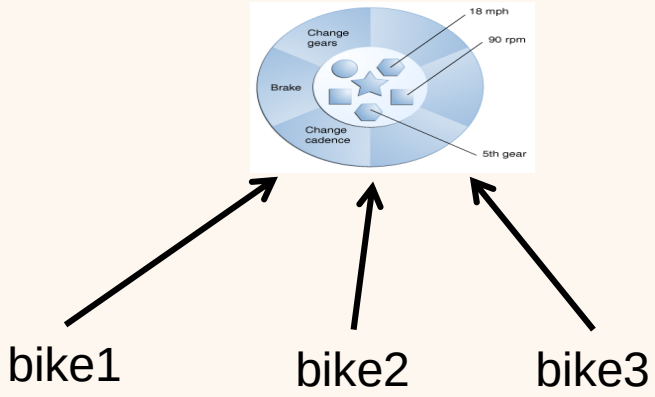


Classes

- Software classes are used to define groups of objects
- Objects of the same class share?
  - *types of members* (i.e., possible *states*)
  - the same *methods* (i.e., *behavior*)
- Objects of the same class (potentially) differ in?
  - *Value of data members*

Reference

- A reference is not an actual object, but something that **points** to an object
- This means that the creation of new references doesn't waste much memory
  - *Bicycle bike1 = **new** Bicycle(1);*
  - *Bicycle bike2 = bike1;*
  - *Bicycle bike3 = bike1;*
  - *…*

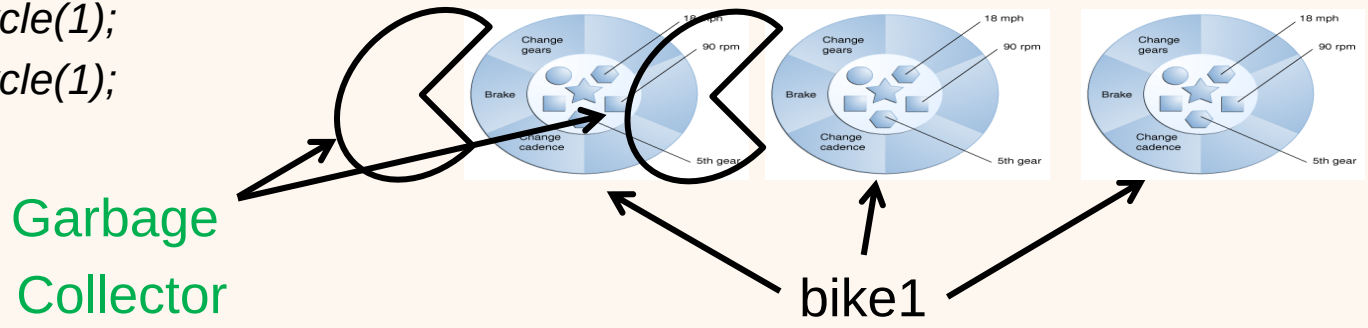


Content Example

*Bicycle bike1 = **new** Bicycle(1);*

*bike1 = **new** Bicycle(1);*

*bike1 = **new** Bicycle(1);*



Reference / Content Question

- What other implications does this distinction have?
  - **Interfaces** and **abstract** **classes** can only appear as references
  - Up-casting:

```
Animal myAnimal = new Dog();
```

  - Generic wildcards:

```
List<?> myList = new LinkedList<String>();
```

  - Program to an interface, not an implementation
    - Exposing the reference type is usually not a problem. The content type should remain hidden whenever possible

The static Modifier

- The **static** modifier associates a variable or method with the class rather than an object

```
public class Dog {
        private static int nDogs = 0;          // Count the number of dogs
        public Dog() {
          Dog.nDogs++;          // Dog.nDogs is increased each time a new Dog is created
          ...
        }
        public static int getDogsCounter() {   return Dog.nDogs;  }
        …
}
```

Minimal API

- When delivering a program, we want to share as few details as possible
  - A **minimal API**
- Most implementation details should not be revealed

Encapsulation

- The grouping of related ideas into **one unit**, which can then be referred to by a **single name**
  - This **unit** is referred to as a **black box**
  - Facilitates human understanding by representing a **conceptually complex concept** as a **single simple idea**

Information Hiding

```
public class A{
        public int a;
        protected int b;
        int c;
        private int d;
        void foo(){
                int e;
        }
}
```

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
*default(=*package*)*	Y	Y	N	N
private	Y	N	N	N

Why not Share?

- The more information we provide about our code, the harder it is for users to learn how to use it
  - Fewer details are easier to grasp
- Clients might **misuse** the API we provide them
- Most importantly, providing details about our code makes it **harder** for us **to modify** it later

Why not Share?

- Where else have we seen this principal?
  - Constants

```
final int a = 5;
```

  - Program to an **interface**, not to an **implementation**

```
Animal myAnimal = new Dog();
```

Lecture 13b: Overview

- Part a: **Introduction to java** (Lectures 1-2)
- Part b: **Polymorphism and Basic Design** (Lectures 3-5)
- Part c: **Core Topics in java** (Lectures 6-7)
- Part d: **Modularity and Advanced Design** (Lectures 8-9)
- Part e: **Advanced Topics** (Lectures 10-13)

Has-a Relation

- The most basic relation between classes is the *has-a* relation (also called **composition**)
- This relation is formed where one object “belongs” to another object
  - A person **has a** name, bicycles **have** wheels, etc.

```
public class Person {
    // A person has a name and a mother (it composes them)
    private String name;
    private Person mother;
    …
}
```

Composition

- What other types of composition have we seen?
  - A Delegates to B
    - A **composes** B and **forwards requests** to the composed instance's **methods**
    - **Code reuse** alternative to **inheritance**
  - A Decorates B
    - A **delegates to** B and **extends** B
    - Add a set of **functionalities** to a set of classes

Is-a Relation

- Consider a class that is a more specific version of an existing class
  - A *student* **is a** *person*
- Represented by inheritance
  - Class *A* inherits (or **extends**, in java) class *B*, if *A is B*

```
public class Student extends Person {
        private int id;
        public void takeExam(…) { … }
        …
}
```

Inheritance, What is it Good for?

- Inheritance represents the *is-a* relation
  - Class *A* should not extend class *B* if *A* is **not** a *B*
- Inheritance allows for **polymorphism**
- Inheritance also serves as a code-reuse mechanism
  - Class *A* can use class *B*'s methods without re-implementing them
- Nevertheless, other code-reuse alternatives exist
  - **Composition**
  - Code-reuse is **not** a good reason to use inheritance

Polymorphism

Example

```
Cow myCow = new Cow();
Dog myDog = new Dog();
Animal myAnimal = myCow;
myAnimal.speak();
myCow.speak();
myCow.getMilk();
myAnimal.getMilk();
mydog.eat();
myCow.eat();
myAnimal.eat();
```

The Cow object takes the form of an animal (**Polymorphism**)

Animals can speak (myAnimal is a cow, so output

A cow is also an animal, so it can speak (“moo”)

Cows give milk

But animals can’t! Even though this object is **actually a cow** (**Compilation Error**)

All animals can *eat* (whether Animal.eat() was overridden or not)

Polymorphism is Useful

```
/** A function that gets an animal argument of any type and makes it speak. */
public void makeAnimalSpeak(      ????      ) {
      ????      .speak();
}

Cow myCow = new Cow();
Dog myDog = new Dog();
Animal myAnimal = new Cow();

makeAnimalSpeak(myCow);

makeAnimalSpeak(myDog);

makeAnimalSpeak(myAnimal);
```

It's the concrete object that counts!

moo

woof

moo

Abstract Classes

- Abstract classes are classes from which we cannot create an instance
  - Defined using the **abstract** keyword
  - May define **zero** or more **abstract** methods

```
public abstract class Animal {
        // An abstract speak method.
        // To be implemented by Animal sub-classes.
         public abstract void speak();
}
Animal animal = new Animal(…);          // Compilation error.
```

Abstract Class – What is it Good for?

- Cases where the top level(s) of our inheritance tree are not concrete classes
  - It makes no sense to create an instance of a general animal
- When we want to force an API on a group of inheriting classes
  - Extending concrete classes **must** implement all **abstract** methods
  - But the parent class cannot provide a reasonable implementation for this API

Interfaces

```
/* An interface for printable objects. */
public interface Printable {
      // A print method
       public void print();
}

public class Document implements Printable {
      // Implementing the Printable.print() method
       public void print() {
                …
      }
      …
}
```

Interface keyword

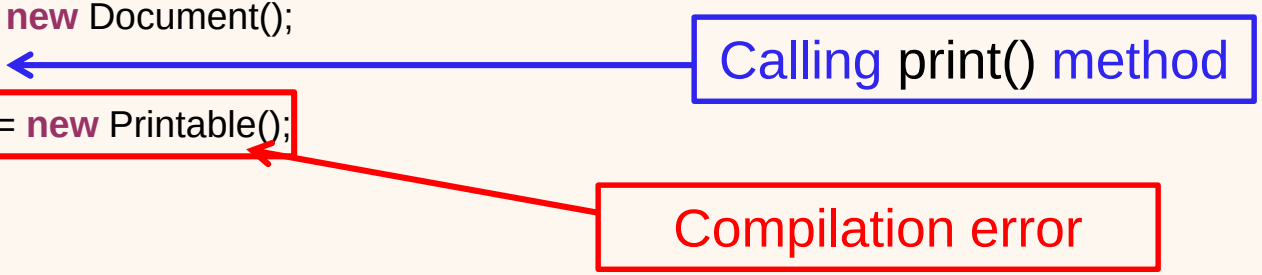
No need for the **abstract** keyword

implements keyword

print() method implementation

Interfaces

```
    ….
    public static void main(String args[]) {
            Printable p = new Document();
            p.print();
            Printable p2 = new Printable();
    }
}
```



Why Use Interfaces?

- Interfaces represent *contracts* that classes accept
  - Unlike classes, they do not represent something in the world, but a **requirement** that is shared among various classes of various types
- Examples:
  - *Printable*: for classes that contain objects that can be printed
  - *Comparable*: for classes that contain objects that can be compared to other objects
  - *Cloneable*: for classes that contain objects that can be cloned
- Interfaces speak about *what*, not about *how*

More on Interfaces

- Interfaces are not part of the class hierarchy
  - Although they work in combination with classes
- In Java, a class can **extend** only one class, but it can **implement** any number of interfaces

  public class MyClass **implements** *MyInterface1, MyInterface2, …*
- Therefore, objects can have **multiple types**
  - The type of **their own class,** the types of **all the classes** they extend (directly and indirectly) and the types of **all the interfaces** that they implement (also, directly and indirectly)

Interfaces and Abstract Classes

- If the *is-a* relation holds between two types, then you should use inheritance (**extends**)
  - A dog **is an** animal, a car **is a** vehicle
- If the common property is more of a contract, or a specific behavior defined by one class and used by another, use an interface (**implements**)
  - Printability, clonability, comparability, …
- In cases of uncertainty, favor **interfaces**

Casting Examples

```
Animal a; Cow c; Dog d;
d = new Dog();          // OK
a = new Cow(5);         // OK (implicit up-casting)
a.speak();              // “moo”
a = d;                  // OK (implicit up-casting)
a.speak();              // “woff”
d = (Dog) a;            // OK  ([explicit] down-casting)
d = new Cow(3);         // Compile-time error (Cow is not a subclass of Dog)
d = a;                  // Compile-time error (implicit down-casting)
c = (Cow) a;            // Run-time error (incompatible down casting)
if (a instanceof Cow) {
      c = (Cow) a;      // OK (though not recommended)
}
```

a	c	d
Dog	Cow?	Dog

Design Patterns Properties

- Describe a proven approach to dealing with common situations in programming / design
- Suggest what to do to obtain elegant, extensible & reusable solutions
- Show, at design time, how to avoid problems that may occur much later
- Are independent of specific contexts or programming languages

Façade: Example
Media Players

```
public class ComplexPlayer {
      public void play(String fileName,
                              int leftvolume,
                              int rightvolume,
                              double bass) { … }
      …
}
```

```
public interface SimplePlayer{
        public void play(String fileName);
}
```

```
public class ComplexPlayerFacade
                  implements SimplePlayer {
      private ComplexPlayer player;
      public ComplexPlayerFacade () {
                  this.player = new ComplexPlayer();
      }
      public void play(String fileName) {
            player.play(fileName, 50, 50, 0.5);
      }
}
```

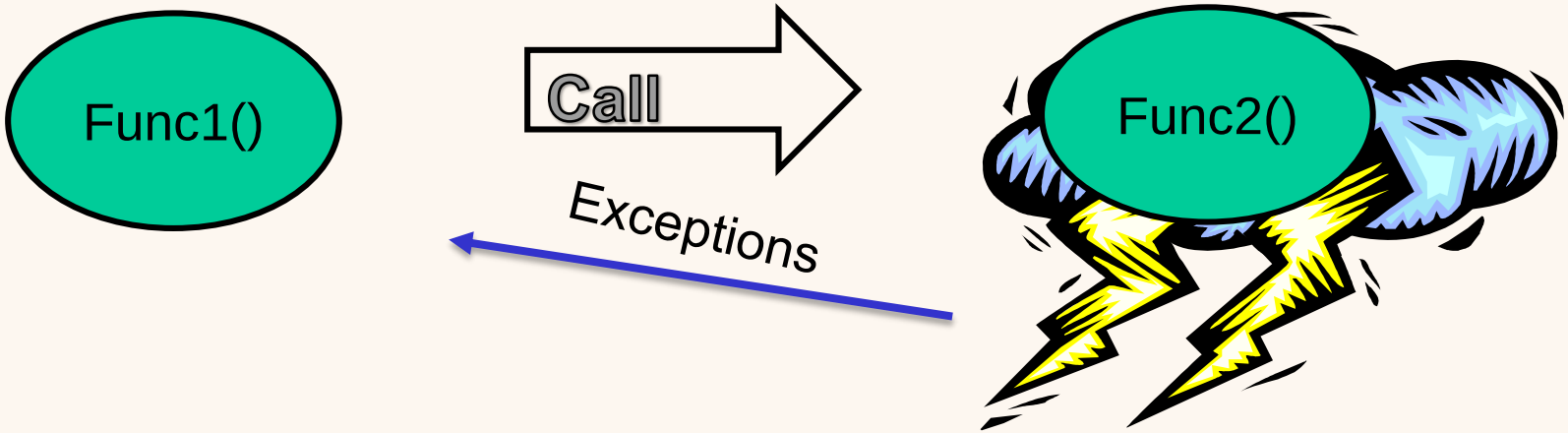
Lecture 13c: Overview

- Part a: **Introduction to java** (Lectures 1-2)
- Part b: **Polymorphism and Basic Design** (Lectures 3-5)
- Part c: **Core Topics in java** (Lectures 6-7)
- Part d: **Modularity and Advanced Design** (Lectures 8-9)
- Part e: **Advanced Topics** (Lectures 10-13)

Summary: Collections

- Generic java structures for holding multiple objects
- Map<K,V> and Collection<E> interfaces
- Interfaces:
  - Ordering, handling duplicates
- Implementations:
  - Complexity requirements
  - Requirements on object methods

Summary: Exceptions



Why Exceptions

- **Separating** error handling code from rest of code
- **Error propagating** up the call stack
  - **foo1() → foo2() → … → foo1000() → exception!**
- **Grouping** together and **differentiating** error types
  - **public class** ListException { … }
  - **public class** EmptyListException **extends** ListException {…}

Lecture 13d: Overview

- Part a: **Introduction to java** (Lectures 1-2)
- Part b: **Polymorphism and Basic Design** (Lectures 3-5)
- Part c: **Core Topics in java** (Lectures 6-7)
- Part d: **Modularity and Advanced Design** (Lectures 8-9)
- Part e: **Advanced Topics** (Lectures 10-13)

Modularity

- A Modular design results in a software that can be broken down to several individual units, denoted **modules**
- Modular programs have several benefits
  - Easy to maintain (debug, update, expand)
  - Allow breaking a complex problem into easier sub-problems
  - Allow to easily divide the project into several team members or groups
- Decomposability, Composability, Understandability, Continuity

Design Principles

- Modules that conform to the **open-closed** principle have two primary attributes:
  - They are "**Open for Extension**"
  - They are "**Closed for Modification**"
- The Single-Choice Principle:
  - If a software system **must** support a set of alternatives, **one and only one** module in the system should know their exhaustive list

The Shape Example

OOP Solution

```
public Drawable[] loadAll (String[] list) {
    Drawable[] drawables = new Drawable[list.length];
    for (int i = 0 ; i < list.length ; ++i) {
        if (list[i].equals("Square")) {
            drawables[i] = new Square();
        } else if (list[i].equals("Circle")) {
            drawables[i] = new Circle();
        } …
    }
    return drawables;
}
```

One method must know all the options.
It **cannot** be closed for changes

The Shape Example

OOP Solution

```
void drawAll(Drawable[] list) {
      for (Drawable drawable: list)
            drawable.draw();
}

void deleteAll (Drawable[] list) {
      for (Drawable drawable: list)
            drawable.delete();
}
```

All other methods don‘t need to know the options.
They are **closed** to changes

More Design Patterns

- Factory
  - An object used to create other objects
- Singleton
  - Ensure a class only has exactly **one instance**, and provide a global point of access to it
- Strategy
  - Define a family of algorithms, encapsulate each one, and make them interchangeable
  - Let the algorithm vary independently from the clients that use it

Summary: streams

- Same framework for different I/O
- Different approaches for Text/Binary


```
Output/InputStream stream = new FileOutput/InputStream(somefile);

byte a = stream.read() /  stream.write(a);

stream.close();
```

Summary: Decorator

- **Objective:** Enhance a family of classes (**streams**) with additional abilities
- **Problems:**
  - Many possible enhancements
  - Many types of classes (**input/output streams**)
- **Solution**: Build class B (BufferedInputStream) that
  - **Extends A** (shares its API)
  - **Contains a component of type A**
    - Constructor of **B** receives an object of type **A** and remembers it

Lecture 13e: Overview

- Part a: **Introduction to java** (Lectures 1-2)
- Part b: **Polymorphism and Basic Design** (Lectures 3-5)
- Part c: **Core Topics in java** (Lectures 6-7)
- Part d: **Modularity and Advanced Design** (Lectures 8-9)
- Part e: **Advanced Topics** (Lectures 10-13)

Summary: Serialization & Cloning

- Serialization:
  - Saving the entire object hierarchy into a stream
  - Each object is saved at most once per stream
- Cloning:
  - Shallow vs. Deep Copy
  - Better alternative: Copy Constructor

Summary: Java Reflection

- Examine the internal structure of any class
  - Including **private** members
- Can extend **flexibility** and **extensibility**
- Violates **encapsulation**!

Overview

Theoretic framework for OOP

Inheritance, Encapsulation, Polymorphism, Abstraction

Java mechanics for OOP

Packages, Classes, Interfaces, Generics

OOP-based design patterns

Façade, Singleton, Strategy, Decorator, …

Advanced programming

Large projects, Streams, Regular expressions, …

GOOD LUCK!