### Object-Oriented Programming

The fundamental unit of programming in Java is the **class**

Classes contain **methods** that perform work

Classes may be **instantiated** to create **objects**; an object is an **instance** of a class

Object-oriented programming separates the notion of “what” is to be done from “how” it is done.

- **“What”**: A class's methods provide a contract via their signatures (i.e. method's parameters types) and their semantics
- **“How”**: Each class may have its own unique implementation of a method

When a method is invoked on an object, its class is examined at runtime to locate the exact code to run (“dynamic dispatch”)

### Fields

Here is a simple class:

```java
class Employee {
    public int id;
    public String name;
    public Employee boss;

    public static int nextId = 0;
}
```

A class's variables are called its **fields**

Every Employee object has its own `id`, `name`, and `boss` – the values of these fields make an Employee unique

A field that is declared **static** is shared among all instances of a class. These fields are called **static fields** or **class variables**.

By convention, class names begin with a capital letter (`Employee`), while field and method names begin with a lowercase letter (`name`)

### Access Control

Fields and methods (“members”) are always available to their declaring class, but you can control other class’s access them with **access control modifiers**:

- **public**: Members declared `public` are accessible anywhere the class is accessible
- **private**: `private` members are only accessible by the class in which they are declared
- **protected**: `protected` members are accessible from direct subclasses and by classes in the same package
- **package**: Members with no declared modifier (default) are accessible only by classes in the same package
Creating Objects

Objects are instantiated with the `new` operator

```java
Employee john = new Employee();
john.id = Employee.nextId++;
john.name = "John";
john.boss = null; // He's the President!

Employee rony = new Employee();
rony.id = Employee.nextId++;
rony.name = "Rony";
rony.boss = john;
```

The `new` operator allocates memory for an object in the JVM's garbage-collected heap

After we create an `Employee` object, we assign values to its fields

Members are referenced using . (e.g. `rony.id`)

The `null` keyword represents no object

- An attempt to reference a `null` object result in a `NullPointerException`

Constructors

`Constructors` are pseudo-methods that initialize a newly-created object. They are invoked when an object is instantiated.

- Constructors have the same name as the class whose instances they initialize
- A constructor may have parameters, but it has no declared return type

```java
class Employee {
    public int id;
    public String name = "<unknown>";
    public Employee boss = null;
    private static int nextId = 0;

    Employee() {
        id = nextId++;
    }
}

Employee john = new Employee();
john.name = "John";
Employee rony = new Employee();
rony.name = "Rony";
rony.boss = john;
```

You may want to create an `Employee` after you know its name and boss:

```java
Employee(String name, Employee boss) {
    this();
    this.name = name;
    this.boss = boss;
}
```

The two-argument constructor invokes the zero-argument constructor by calling `this()`

The reference `this.name` refers to the `name` field of the current object and not the parameter `name`

```java
Employee john = new Employee("John", null);
Employee rony = new Employee("Rony", john);
```

Notice that the code fragment no longer needs to know about the `name` and `boss` fields.

Methods

`Methods` contain code that often manipulates an object's state

Public fields are usually not a good idea – you should expose behavior, not state – have methods do the work

Methods are `invoked` on references to objects using the . operator:

```java
receiver . method (parameters)
```

Methods have parameters each with its own type as well as a return type

```java
public String toString() {
    String desc = id + " (" + name + ")";
    if (boss != null)
        desc += " boss " + boss.toString();
    return desc;
}
```

Invoking `rony.toString()`:

```
1 (Rony) boss 0 (John)
```
Parameter Passing

Java method calls are “pass by value” – any changes made to a parameter by the callee will not be reflected in the caller.

```java
public class PassByValue {
    private static void doubled(int i) {
        i = i * 2;
        System.out.println("Doubled: " + i);
    }

    public static void main(String[] args) {
        int i = 27;
        System.out.println("Before: " + i);
        doubled(i);
        System.out.println("After: " + i);
    }
}
```

This code will print:

```
Before: 27
Doubled: 54
After: 27
```

However, you can mutate the state of an object that is passed by value.

```java
public class ChangeName {
    private static void changeName(Employee emp) {
        if (emp.boss == null) {
            emp.name = "President";
            emp = new Employee("Fred", emp);
            System.out.println("Changed to: " + emp);
        }
    }

    public static void main(String[] args) {
        Employee john = new Employee("John", null);
        System.out.println("Before: " + john);
        changeName(john);
        System.out.println("After: " + john);
    }
}
```

This program prints out:

```
Before: 0 (John)
Changed to: 1 (Fred) boss 0 (President)
After: 0 (President)
```

Accessing private Data Using Methods

You can use methods control how fields are manipulated.

Make the `id` field private

```java
public class Employee {
    private int id;
    public String name = "<unknown>";
    public Employee boss = null;

    private static int nextId = 0;

    Employee() {
        id = nextId++;
    }

    public int getId() {
        return this.id;
    }
}
```

Now, the only way to set `id` is through the constructor.

Because it is used to access data, `getId` is called an accessor method.

Overloading Methods

A method’s signature is comprised of the method’s number and types of parameters.

Method overloading involves having multiple methods with the same name, but different signatures.

```java
public void setName(String name) {
    this.name = name;
}

public void setName(String firstName, String lastName) {
    this.name = firstName + " " + lastName;
}
```

The `setName` method is overloaded.

SmallTalkers like to overload methods thusly:

```java
public Employee boss() {
    return this.boss;
}

public void boss(Employee boss) {
    this.boss = boss;
}
```

I prefer “get” and “set” accessors and mutators.
Static Fields

A static member is associated with a class instead of an object.

For instance, there is only one nextId variable for all Employees (like a “global variable”).

A class’s static fields are initialized before any static field is referenced or any method is run.

Static fields may also be initialized in a static initialization block.

```java
public class Day {
    public static String[] daysOfWeek;
    static {
        daysOfWeek = new String[7];
        daysOfWeek[0] = "Sunday";
        daysOfWeek[1] = "Monday";
        daysOfWeek[2] = "Tuesday";
        // ...
    }
}
```

String monday = Day.daysOfWeek[1];

---

Static Methods

A static method is invoked with respect to an entire class, not just an instance of a class.

Static methods are also called class methods.

Static methods have no this variable, so static methods can only access static variables and can only invoke static methods of its declaring class.

You usually invoke a class method with

```
className.methodName(parameters)
```

```java
public static String getDay(int i) {
    return daysOfWeek[i];
}
```

String monday = Day.getDay(1);

Recall that the main method is static.

---

Static import

Prior to J2SE 1.5, static fields and methods had to be qualified by their class name:

- Arithmetic functions were more verbose than in languages like C and PASCAL: `Math.abs(4)`
- The long names of constants such as `BorderLayout.CENTER` led to the “antipattern” of creating interfaces consisting solely of constants.

In J2SE 1.5, you can use the `import static` statement:

```java
package edu.pdx.cs410J.j2se15;
import static java.lang.Integer.*;
import static java.lang.System.*;
```

```java
public class StaticImports {
    public static void main(String[] args) {
        int sum = 0;
        for (int i = 0; i < args.length; i++) {
            // Integer.parseInt()
            sum += parseInt(args[i]);
        }
        // System.out
        out.println("Sum is " + sum);
        // Integer.MAX_VALUE
        out.println("MAX_INT is " + MAX_VALUE);
    }
}
```

*I’m still not convinced that static imports really make the code easier to read.*
_objects are allocated using `new`, but are implicitly deallocated by the garbage collector.

An object is considered garbage when it meets all of the following criteria:

- It is not referenced by a static field
- It is not referenced by a variable in a currently executing method
- It is not referenced by a _live_ (non-garbage) object

The programmer cannot destroy an object – there is no “delete” nor “free”

- No dangling references
- Makes the language a lot safer
- You can still have “memory leaks”, though

The object heap in a Java VM might look like this:

```
public class Foo {
  static Bar bar;
}
```

Currently executing methods on the stack and static fields keep objects alive.

When the currently executing method returns, the objects it referred to become candidates for garbage collection.

Finally, the objects that are no longer references are deallocated and recycled for future use.

The garbage collection starts with live objects and follows their object references in a _transitive closure_.

The object heap in a Java VM might look like this:
The finalize Method

An object's finalize method is invoked just before it is garbage collected.

The finalize method should release any system resources (such as file handles or sockets) that the object might have open.

```java
import java.net.Socket;

public class Finalize {
    private Socket socket;

    public Finalize() {
        socket = openSocket();
    }

    protected void finalize() throws Throwable {
        socket.close();
        super.finalize();
    }
}
```

finalize should always invoke its superclass's finalize method.

It is not recommended that you “resurrect” objects in their finalize methods.

Extending Classes

One of the main advantages of object-oriented programming is code reuse.

One way to reuse classes is to extend their functionality by subclassing them.

A class's methods and public fields specify a “contract” that it provides.

When you extend a class, you add new functionality to its contract and you also inherit its existing contract.

A subclass may change the implementation of its superclass's contract, but the semantics of the contract should not change.

A subclass can be used wherever its superclass can be used. This property is called polymorphism.

java.lang.Object is the root of Java's class hierarchy.

Everything “is an” Object

```java
String s = "I'm a String!";
Object o = s;
```

Extending Classes

Unless otherwise specified, a class directly subclasses Object. (This was the case for Employee.)

We use the extends keyword to indicate a class's superclass.

```java
class Consultant extends Employee {
    private String mobile;
    private String pager;

    public Consultant(String name, Employee boss, String mobile, String pager) {
        super(name, boss);
        this.mobile = mobile;
        this.pager = pager;
    }

    public String toString() {
        String desc = this.name + " M: " + mobile + " P: " + pager;
        return desc;
    }
}
```

Consultant declares two fields, mobile and pager, in addition to the fields it inherits from Employee, name and boss.

Consultant cannot access Employee's id field because it is private.

Consultant's constructor must invoke its superclass's constructor before performing any other actions:

```java
super(name, boss);
```

invokes Employee's two-argument constructor that initializes the name and boss fields inherited by the Consultant.

* If a constructor does not explicitly invoke an inherited constructor, the superclass's zero-argument constructor will be invoked.*

In the toString method this.name refers to the name field inherited from Employee.

*If the superclass does not provide a zero-argument constructor, the subclass will not compile!*

Looking at the Consultant class

Consultant declares two fields, mobile and pager, in addition to the fields it inherits from Employee, name and boss.

Consultant cannot access Employee's id field because it is private.

Consultant's constructor must invoke its superclass's constructor before performing any other actions:

```java
super(name, boss);
```

invokes Employee's two-argument constructor that initializes the name and boss fields inherited by the Consultant.

* If a constructor does not explicitly invoke an inherited constructor, the superclass's zero-argument constructor will be invoked.*

In the toString method this.name refers to the name field inherited from Employee.

*If the superclass does not provide a zero-argument constructor, the subclass will not compile!*

Note that concatenating an object to a String invokes its toString method.

Looking at the Consultant class

Consultant declares two fields, mobile and pager, in addition to the fields it inherits from Employee, name and boss.

Consultant cannot access Employee's id field because it is private.

Consultant's constructor must invoke its superclass's constructor before performing any other actions:

```java
super(name, boss);
```

invokes Employee's two-argument constructor that initializes the name and boss fields inherited by the Consultant.

* If a constructor does not explicitly invoke an inherited constructor, the superclass's zero-argument constructor will be invoked.*

In the toString method this.name refers to the name field inherited from Employee.

*If the superclass does not provide a zero-argument constructor, the subclass will not compile!*

Note that concatenating an object to a String invokes its toString method.
Overriding Methods

Consultant overrides Employee's toString method.

Overriding replaces the implementation of a superclass's method with the subclass's. The signature of the overriding method must match that of the overridden class.

The access specifier of an overriding method may not restrict access to the method. That is, we could not make toString protected.

Fields cannot be overridden, but they can be hidden

• Consultant could declare a name field that would hide the field declared in Employee

• Any references to name would refer to the name field declared in Consultant

• Note that name in methods inherited from Employee would still refer to the name field in Employee

The super keyword

The super keyword treats the current instance (i.e. the object referred to by the this variable) as an instance of its superclass.

Consultant could have used super.name to refer to the name field declared in its superclass.

It could also use super to invoke methods of its superclass

```java
public String toString {
    String desc = super.toString();
    desc += " M: " + mobile;
    desc += " P: " + pager;
    return desc;
}
```

Note that super can be used to reference protected members of the superclass

25

final Classes and Methods

Marking a method as final prevents any subclass from overriding it.

Marking a class as final prevents it from being extended.

Final classes and methods provide security in that they ensure that behavior will not change.

• Someone can't override a final validatePassword method to always return true.

Final classes and methods may also provide an opportunity for optimization.

• You don't need to perform a dynamic dispatch of a final method because you know that none of its subclasses can override it.

The value of fields that are declared to be final cannot be changed.

• Can only be assigned to in constructors.

• A field that is final and static is a considered to be constant.

```java
public class Circle {
    public static final double PI = 3.14159;
    private double radius;
    public Circle(double radius) {
        this.radius = radius;
    }
    public double getCircumference() {
        return 2.0 * PI * this.radius;
    }
}
```

The names of constants are usually in ALL_CAPS.

27

final Fields

28
Methods Provided by the Object Class

Every class inherits the following methods from Object:

- **public boolean equals(Object obj)**
  - Compares the receiver object to another object
  - Value equality (as opposed to == and !=)

- **public int hashCode()**
  - Returns a hash code for the object (used when storing the object in hash tables)

- **protected Object clone() throws CloneNotSupportedException**
  - Returns a copy (a “clone”) of the receiver object

- **public final Class getClass()**
  - Returns the instance of java.lang.Class that represents the class of the receiver object

- **protected void finalize() throws Throwable**

abstract Classes and Methods

Abstract classes allow you to delegate the implementation of a class or method to the subclass.

Makes sense when some behavior is true for most instances, but other behavior is specific to a given subclass.

Abstract methods are denoted with the abstract keyword and have no method body.

Abstract classes are denoted with the abstract keyword.

- Abstract classes cannot be instantiated, although they may declare constructors
- Any class that has an abstract method, must also be declared abstract

A class hierarchy example: animals

We will use the following class hierarchy to demonstrate classes, interfaces, inheritance and instance methods:

```
package edu.pdx.cs410J.lang;

/**
 * This class is the base class in our animal hierarchy. Every animal has a name and it makes a sound.
 */
public abstract class Animal {
    protected String name;

    /**
     * Returns the name of this animal.
     */
    public final String getName() {
        return this.name;
    }

    /**
     * Returns the sound that this animal makes.
     */
    public abstract String says();

    public String toString() {
        return getName() + " says " + says();
    }
}
```

The Animal class

Abstract classes are italicized

Concrete (non-abstract) classes are in boldface
package edu.pdx.cs410J.lang;

/**
 * This class represents all mammals.
 */
public abstract class Mammal extends Animal {
    /** All mammals can produce milk */
    public void giveMilk() {}
}

package edu.pdx.cs410J.lang;
/**
 * This class represents an insect.
 */
public abstract class Insect extends Animal {
    /** By default, all insects say "Buzz"
     */
    public String says() {
        return "Buzz";
    }
}

package edu.pdx.cs410J.lang;
/**
 * This class represents a cow.
 */
public class Cow extends Mammal {
    public Cow(String name) {
        this.name = name;
    }
    public String says() { return "Moo"; }
}

package edu.pdx.cs410J.lang;
/**
 * This class represents a human being.
 */
public class Human extends Mammal {
    public Human(String name) {
        this.name = name;
    }
    public String says() { return "Hello"; }
}

package edu.pdx.cs410J.lang;
public class SayWhat {
    private static Human human;
    private static Cow cow;
    private static Ant ant;

    /** Prints an animal’s name and what it says. */
    private static void saysWhat(Animal animal) {
        System.out.println(animal.getName() + " says " + animal.says());
    }

    /**
     * This main method creates a number of animals and prints out their names and what they say.
     */
    public static void main(String[] args) {
        human = new Human("Dave");
        cow = new Cow("Bessy");
        ant = new Ant("Arthur");

        saysWhat(human);
        saysWhat(cow);
        saysWhat(ant);
    }
}

Compile source code
$ javac -classpath ~/classes -d ~/classes SayWhat.java

Run Java program
$ java -cp ~/classes edu.---.SayWhat
Dave says "Hello"
Bessy says "Moo"
Arthur says "Buzz"

• The argument to the saysWhat method was an Animal, but the choice of which says method to call was delayed until runtime
• All subclasses of Animal invoke the same getName method because it was not overridden
• Even though we appear to invoke Animal’s abstract says method, we really invoke the subclass’s concrete method
• Understanding this abstraction is the key to object-oriented programming
Do pigs fly?

Some animals in our hierarchy fly. How should we represent this?

We could use a class, but where would it fit in our hierarchy?

Some insects fly (Bees) and some birds (Turkeys) don’t fly

Solution: Use an interface

Um, what’s an interface?

An interface is like a class, but it contains only method declarations

- Used to declare methods that a class should implement
- Does not specify how those methods are implemented
- A class uses the implements keyword to denote that it implements the methods of an interface
- A class may implement more than one interface: multiple inheritance of behavior
- An interface may extend another interface
- All methods of an interface are implicitly public
- All fields of an interface are implicitly public, static, and final (constants)
- An interface is a type just like a class (i.e. you can have variables of an interface type)

The Files interface

In file edu/pdx/cs410J/lang/Flies.java:

```java
package edu.pdx.cs410J.lang;
/**
 * This interface is implemented by all animals that can fly.
 */
public interface Flies {
    public void fly();
}

Every class that implements Flies must have a fly method (or be declared abstract)
```

Ducks and Bees fly

In file edu/pdx/cs410J/lang/Flies.java:

```java
package edu.pdx.cs410J.lang;
/**
 * This class represents a duck. Ducks fly. */
public class Duck extends Bird implements Flies {
    public Duck(String name) {
        this.name = name;
    }
    public String says() { return "Quack"; }
    public void fly() {
        System.out.println("I'm flying");
    }
}
```

In file edu/pdx/cs410J/lang/Flies.java:

```java
package edu.pdx.cs410J.lang;
/**
 * This class represents a bee. Bees can fly. */
public class Bee extends Insect implements Flies {
    public Bee(String name) {
        this.name = name;
    }
    public void fly() {
        System.out.println("I'm flying");
    }
}
```
Performing a type test: `instanceof`

How can we tell whether or not an object is of a given type?

The `instanceof` operator checks the type of an object

```java
/**
 * If possible, milk an animal
 */
void milk(Animal animal) {
    if (animal instanceof Mammal) {
        Mammal mammal = (Mammal) animal;
        mammal.giveMilk();
    }
}
```

Only Mammals have the `giveMilk` method

We cast the animal to a Mammal after we've done the type test. This way, we know we have an instance of Mammal.

If we attempt to cast an object to a type that it is not, a `ClassCastException` is thrown.

If we didn't cast `animal` to a Mammal, the code wouldn't compile because Animal does not have a `giveMilk` method. (Java is “strongly typed”)

---

Example program using the Flies interface

```java
package edu.pdx.cs410J.lang;

public class DoIFly {
    /** Prints out whether or not an animal can fly. */
    private static void doIFly(Animal animal) {
        boolean iFly = (animal instanceof Flies);
        System.out.print("Does " + animal.getName() + " fly? ");
        System.out.println((iFly ? "Yes." : "No."));
    }

    /**
     * This main program creates several animals and
     * then prints out whether or not they can fly.
     */
    public static void main(String[] args) {
        doIFly(new Cow("Bessy"));
        doIFly(new Bee("Buzz"));
        doIFly(new Turkey("Tom"));
    }
}
```

Compile the source code:

```bash
$ cd edu/pdx/cs410J/lang
$ javac -classpath ~/classes -d ~/classes \DoIFly.java
```

Run the program:

```bash
$ java -cp ~/classes \edu.pdx.cs410J.java.DoIFly
Does Bessy fly? No.
Does Buzz fly? Yes.
Does Tom fly? No.
```

Once again, notice how decisions are left until runtime.

---

Okay, so what?

Object-oriented programming allows you to work with data abstractly

All you know is that you have an Animal, you don’t care what kind of Animal it is

“Program to the interface”

Facilitates code enhancement and reuse
### Differences Between Java and C++

Java does not have pointers! You can’t perform pointer arithmetic.

Java uses some different terminology
- “Methods” instead of “member functions”
- “Fields” instead of “member variables”
- “Instance methods” instead of “virtual functions”

Java does not have template classes (yet)

By default, instance methods are virtual

Java objects cannot be allocated on the runtime stack

Java does not have multiple class inheritance

No “friend” classes

No explicit memory deallocation

! operator only works on booleans

A class’s declaration and definition are in the same file (no “header files”)

All classes inherit from Object

### Summary

Object-oriented programming separates data’s interface from its implementation

Java classes have fields that hold data and methods that perform work

Classes, fields, and methods have access modifiers that encapsulate data and functionality

Static members are associated with classes and instance members are associated with objects

A class’s functionality is extended through inheritance and by overriding its methods