Programming in Java

7. Inheritance

Introduction

- The focus of this module is the concepts of inheritance, abstraction and how these are implemented in Java
- There a more background material than we can cover in class
  - There is additional content in the extras folder in the lab
- The main Java related topics to be covered are:
  - The difference between implementing specialization versus generalization inheritance
  - Inheritance hierarchies of Java classes
  - Abstract classes
  - Interfaces as types

Domain versus Design Classes

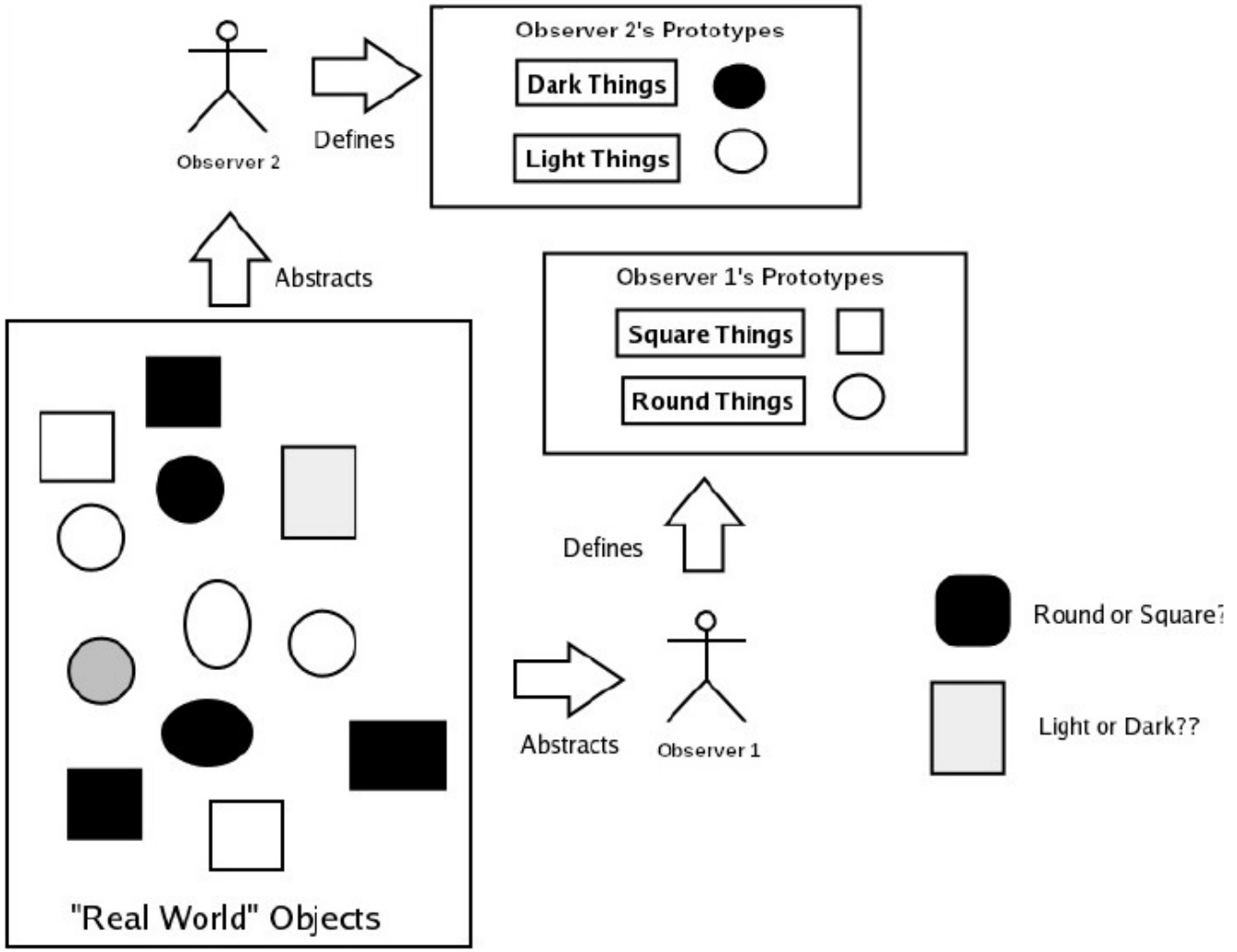
- Domain Classes
  - These correspond to the types that have conceptual reality in the real-world application domain.
  - Domain classes use prototypes that we have to discover by investigation and analysis
  - Domain definitions tend to be fluid and fuzzy
- Design Classes
  - These are the classes we define to build our system
  - These are the classes that exist in the software and do not necessarily copy the domain classes in our problem domain
  - However, the choice of design classes (Java packages and classes) are guided by an understanding of the domain
  - Uur design classes resemble domain classes, they don’t slavishly copy them.

The Chen Hierarchy

- Mental Models
  - Information concerning entities and relationships which exist in our minds
  - This tends to be idiosyncratic and varies from person to person
  - Translating these into code is usually a really bad idea
  - Types are represented as prototypes
- Information structure
  - Organization of information in which entities and relationships are represented by data
  - Agreed upon standard models of the world by user communities
    - *“When we say ‘customer’ we mean….”*
  - Defined by clear and unambiguous predicates and value sets
- Data Structures
  - The implementation of the information structure in something like a relational table or Java class
- Chen’s paper is in the extras folder

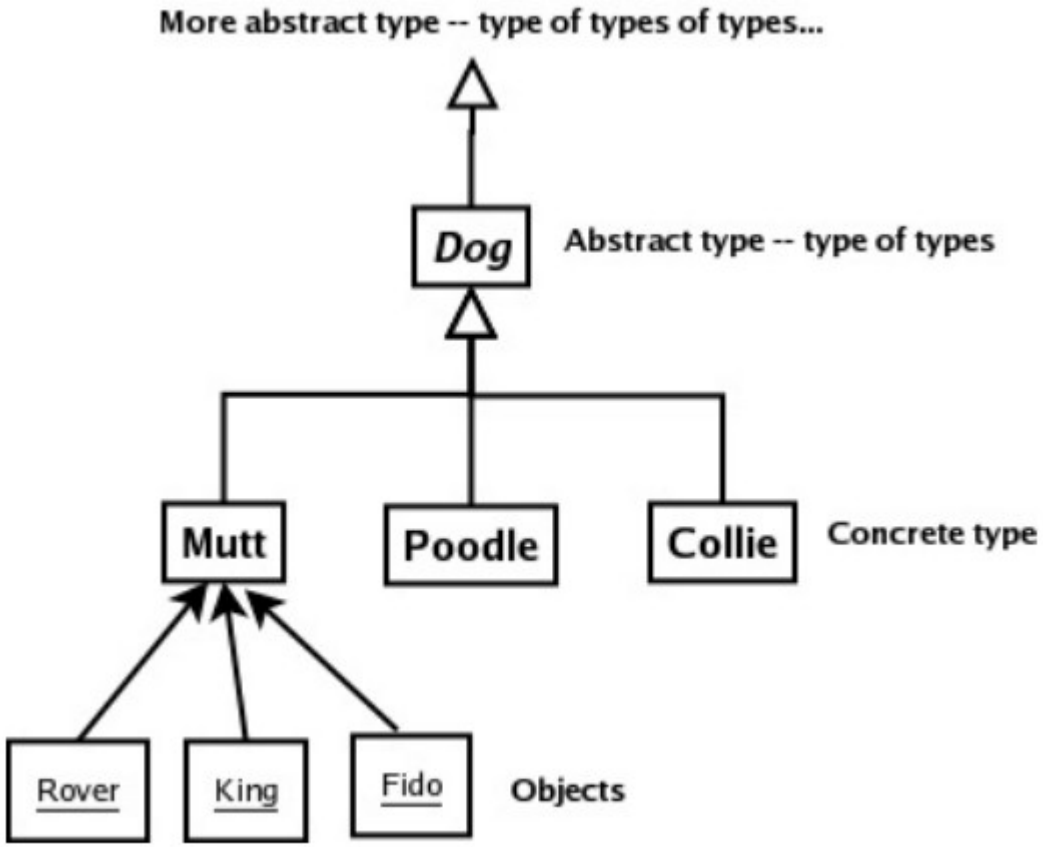
Abstraction

- We perform abstraction, also called generalization, based on three kinds of similarity among objects
  - **Appearance**: Objects that are perceptually similar. e.g. trees, rocks, circles, bangs, stinks
  - **Interaction**: Objects that we interact with in similar ways. e.g. tools, vehicles, food
  - **Relationship**: Objects that exist in similar relationships with other objects. e.g. employees, parents, furniture



Inheritance and Abstract Types

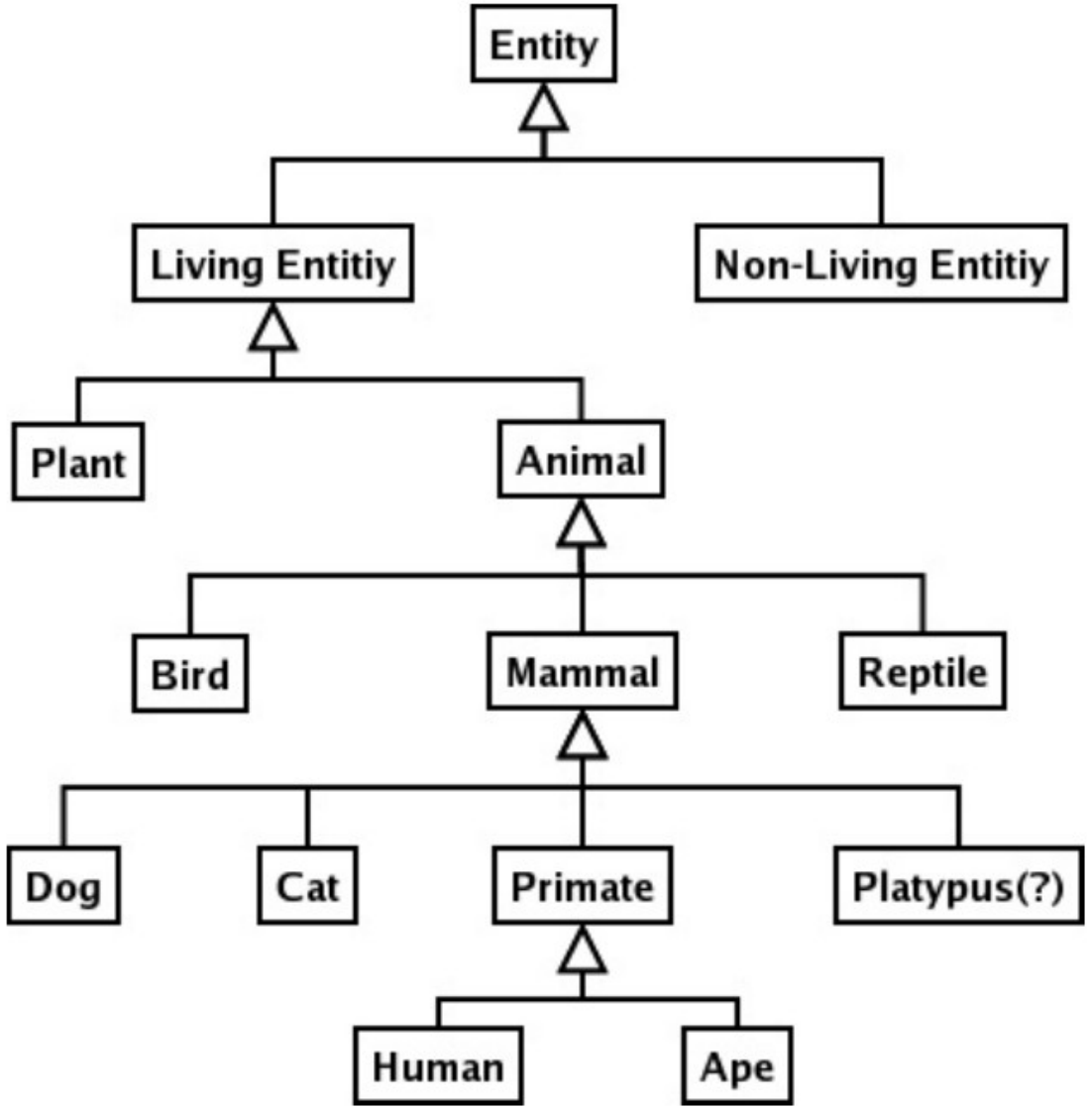
- If a class is an object then we can group classes into classes-of-classes which we call abstract classes
  - This is the start of our abstraction hierarchy
- How do we know something is an abstract class?
  - Because the class description is generally not complete enough for us to specify a well-formed object of that type.
  - Or in plain English objects of only that class just don't exist



Abstract Types

- Abstract classes, and prototypes in general, are often used when recording business rules or system knowledge in a form like a script or user story
  - People tend to store and communicate information as scenarios
  - Details not germane to the point of the story are abstracted away
    - To get money out of an account, a customer goes to an ATM, swipes their card, enters their PIN, selects withdraw, selects the account and amount, then takes their cash and receipt.
  - We don’t need to know concrete details like how much they withdrew, where the ATM was, who the customer is – the focus in on the process
- Constructing these abstraction hierarchies allows for efficient information processing and inference

Abstract Types



Generalization and Specialization

- Generalization refers to the process just described
  - We group a set of classes into a super-type or abstract class
  - We can continue to do this to create a hierarchy where only the lowest level are concrete classes
- Specialization is where we take an existing concrete class
  - Create a new sub-type with additional functionality or specialized use
  - There are no abstract types involved
  - For example:
    - Service dog is a specialization of the concrete class dog
    - Dress shoes is a specialization of shoes
    - Drive through ATM is a specialization of ATM

Specialization in Java

- In Java, we use the ‘extends’ keyword to indicate inheritance
- In the example
  - Animal is a class
  - Dog extends or is a specialized subclass
  - Dog has access to all of the non-private data and methods in Animal
- When creating a Dog
  - An Animal is first created
  - Then the extra stuff defined in Dog is added

```java
// Parent class (also called the superclass)
class Animal {
    void speak() {
        System.out.println("The animal makes a sound");
    }
}

// Child class (inherits from Animal)
class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks");
    }
}

// Main class to test inheritance
public class InheritanceExample {
    public static void main(String[] args) {
        Dog myDog = new Dog();

        myDog.speak();  // Inherited from Animal
        myDog.bark();   // Defined in Dog
    }
}
```

Constructors

- We may need to call a constructor on the super class
  - We do this by creating the same constructor in the sub class
  - Then “relaying” any values to the super class constructor using th e super() method
- This is the only way

```java
// Parent class with a constructor that takes a parameter
class Animal {
    Animal(String name) {
        System.out.println("Animal constructor called. Name: " + name);
    }
}

// Child class with its own constructor that also takes a parameter
class Dog extends Animal {
    Dog(String dogName) {
        super(dogName);  // Call parent constructor with the parameter
        System.out.println("Dog constructor called. Dog name: " + dogName);
    }
}

// Main class to run the example
public class SuperConstructorWithParam {
    public static void main(String[] args) {
        Dog myDog = new Dog("Rex");
    }
}
```


Lab 7-1
Specialization

Overriding Methods

- When we want to replace a super class method with a sub class version
  - This is called overriding
  - The signatures of the method in both classes have to be the same
  - Java will use the sub class version although the super class version is still there
  - We use the @Override annotation so the compiler can validate we did it right

```java
// Parent class
class Animal {
    void speak() {
        System.out.println("The animal makes a sound");
    }
}

// Child class that overrides the speak() method
class Dog extends Animal {
    @Override
    void speak() {
        System.out.println("The dog barks");
    }
}
```

Extending Methods

- We often want to just add something to the super class method
  - We call the super class version using superclass.method()
  - Remember the super class version is still there
  - Because we are not replacing the super class method, just adding functionality to it, this is often called extending a method.

```java
// Parent class
class Animal {
    void speak() {
        System.out.println("The animal makes a sound");
    }
}

// Child class overrides speak() and calls super.speak()
class Dog extends Animal {
    @Override
    void speak() {
        super.speak();  // Call the superclass version first
        System.out.println("The dog barks");
    }
}
```

Lab 7-2

Method Overriding

UpCasting

- Upcasting means treating a subclass object as if it were a superclass object.
  - For example, we can refer to a Dog as either an Animal or a Dog
  - We can use a parent class variable hold a reference to a child class object
  - But you can’t all any methods on an upcase that are only in the subclass
  - Java can’t predict that at runtime, the variable might not contain a Dog so it disallows them

```java
// Superclass with no methods
class Animal {
    // No methods or fields
}

// Subclass with its own method
class Dog extends Animal {
    void wagTail() {
        System.out.println("The dog wags its tail");
    }
}

// Main class to demonstrate upcasting
public class UpcastingDemo {
    public static void main(String[] args) {
        Dog myDog = new Dog();           // Create a Dog object
        Animal animalRef = myDog;        // Upcasting: Dog object as Animal reference

        // animalRef.wagTail();          // This line would cause a compile-time error
    }
}
```

DownCasting

- Reverses the effect of an upcast
- In the code
  - We upcast the Dog to an Animal when we create it
  - Java will not allow us to assign it to a Dog type variable
  - It can’t read back over the code and figure out this is okay
  - We cast it to a Dog when we assign it
  - We are essentially telling the compiler “Don’t worry, I am promising that really is a Dog object”
  - Of course, if it isn’t, a run time error will happen

```
public class DowncastingDemo {
    public static void main(String[] args) {
        Animal animalRef = new Dog();    // Upcasting implicitly
        // animalRef.wagTail();          // Not allowed: Animal reference

        Dog dogRef = (Dog) animalRef;    // Downcasting back to Dog
        dogRef.wagTail();                // Now allowed
    }
}
```

Lab 7-3

Upcasting and Downcasting

Generalization

- This is when we decide that a group of object form a natural collection
  - We create a generalized version that includes all of them
- For example:
  - We have the types Dog and Cat
  - They have a lot in common so we create a new super class called Pet that they both are sub classes of

```java
// Superclass
class Pet {
    // No methods or fields for simplicity
}

// Subclass Dog
class Dog extends Pet {
    void bark() {
        System.out.println("The dog barks");
    }
}

// Subclass Cat
class Cat extends Pet {
    void meow() {
        System.out.println("The cat meows");
    }
}
```

Generalization

- We may move common methods or data from the sub classes to the super class
  - We provide a default version of the method in the super class
  - Then override it in the sub classes

```java
// Superclass
class Pet {
    void speak() {
        System.out.println("The pet makes a sound");
    }
}

// Subclass Dog
class Dog extends Pet {
    @Override
    void speak() {
        System.out.println("The dog barks");
    }

    void fetch() {
        System.out.println("The dog fetches a stick");
    }
}

// Subclass Cat
class Cat extends Pet {
    @Override
    void speak() {
        System.out.println("The cat meows");
    }

    void scratch() {
        System.out.println("The cat scratches the post");
    }
}
```

Generalization

- We do this to create more efficient collections
  - Like this array of pets
  - For common methods, Java calls the right version at runtime
  - Based on the type of object not the type of variable
- Because we are doing so much upcasting
  - We can use the ‘instanceOf’ operator to see what the actual type is of the object in the variable
  - The code is typical way of doing this in Java

```
public class PetArrayExample {
    public static void main(String[] args) {
        // Create an array of Pet references
        Pet[] pets = new Pet[3];

        pets[0] = new Dog();  // Upcast to Pet
        pets[1] = new Cat();  // Upcast to Pet
        pets[2] = new Dog();  // Another Dog

        // Loop through and call the speak method
        for (Pet pet : pets) {
            pet.speak();  // Polymorphic call
        }

        // Example of downcasting within the loop
        for (Pet pet : pets) {
            if (pet instanceof Dog) {
                ((Dog) pet).fetch(); // Downcast to Dog
            } else if (pet instanceof Cat) {
                ((Cat) pet).scratch(); // Downcast to Cat
            }
        }
    }
}
```

Abstract Classes

- Pet is an example of an abstract class
  - This is one that we never are going to instantiate into an object
  - It's more of a building block for its sub classes
  - We can enforce this by making the class Abstract
  - This means we can only create sub classes from it, we can never create an object of that type
- Non-abstract classes are called concrete classes

```java
// Abstract superclass
abstract class Pet {
    // Abstract method to be implemented by all subclasses
    abstract void speak();
}

// Subclass Dog
class Dog extends Pet {
    @Override
    void speak() {
        System.out.println("The dog barks");
    }

    void fetch() {
        System.out.println("The dog fetches a stick");
    }
}

// Subclass Cat
class Cat extends Pet {
    @Override
    void speak() {
        System.out.println("The cat meows");
    }

    void scratch() {
        System.out.println("The cat scratches the post");
    }
}
```

Abstract Classes

- We can also have abstract methods
  - These are methods without bodies
  - Java will not instantiate sub classes unless they override the abstract class with an implementation
  - Abstract classes can have both abstract and non-abstract methods together
  - Like when providing a default implementation for a method
  - If a class has even one abstract method, it becomes an abstract class

```
// Abstract superclass
abstract class Pet {
    // Abstract method to be implemented by all subclasses
    abstract void speak();
}

// Subclass Dog
class Dog extends Pet {
    @Override
    void speak() {
        System.out.println("The dog barks");
    }

    void fetch() {
        System.out.println("The dog fetches a stick");
    }
}

// Subclass Cat
class Cat extends Pet {
    @Override
    void speak() {
        System.out.println("The cat meows");
    }

    void scratch() {
        System.out.println("The cat scratches the post");
    }
}
```

Marker Classes

- These are empty abstract classes that are used only to create an inheritance hierarchy
  - Has the effect of tagging a bunch of classes as being these same type
  - The classes do not need to be related in any other way

```
// Empty abstract superclass
abstract class Pet {
    // No fields or methods
}

// Dog class
class Dog extends Pet {
    void fetch() {
        System.out.println("The dog fetches a stick");
    }
}

// Cat class
class Cat extends Pet {
    void scratch() {
        System.out.println("The cat scratches the post");
    }
}
```


Lab 7-4
Generalization

The Object Class

- In Java, Object is the root class of the entire class hierarchy.
- Every class in Java implicitly extends Object, either directly or indirectly.
- All classes inherit methods from Object, such as toString(), equals(), hashCode(), getClass(), and clone().
- You can use an Object reference to hold any type of object
- The documentation for Object
  - https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/lang/Object.html

Lab 7-5

The Object Class

Interfaces

- Java is a single inheritance language
  - That means each class can have only one *immediate* super class
  - You can’t do “class x extends y,x {}” like in C++
  - A Java Interface allows us to get around that
- An interface is an
  - An abstract class
  - Contains abstract methods
  - Can contain public static final constants
  - It can also contain default implementations and static methods but these are less commonly used

Interfaces

- Interfaces are defined as shown
  - These can be thought of as light-weight abstract classes
- Abstract classes
  - Are used when want to provide a part of class definition with methods and data that are added to in sub classes
- Interfaces
  - Assume each class should implement the specified methods in a unique way

```
interface Flyable {
    int MAX_ALTITUDE = 10000; // public static final by default

    void fly();  // public abstract by default
}

// Second interface
interface Swimmable {
    int MAX_DEPTH = 500; // public static final by default

    void swim();  // public abstract by default
}

// Class implementing both interfaces
class Duck implements Flyable, Swimmable {
    @Override
    public void fly() {
        System.out.println("The duck flies up to " + MAX_ALTITUDE + " feet.");
    }

    @Override
    public void swim() {
        System.out.println("The duck swims down to " + MAX_DEPTH + " feet.");
    }
}
```

Interfaces

- Classes “implement” interfaces
  - They have to override each method in the interface with an implementation
  - A class can implement any number of interfaces
- An interface is a type
  - It defines a bundle of functionality that needs to be implemented for a class to be of that type
  - This is extensively used in Java
  - Interfaces can inherit from other interfaces just like classes do

```
// Class implementing both interfaces
class Duck implements Flyable, Swimmable {
    @Override
    public void fly() {
        System.out.println("The duck flies up to " + MAX_ALTITUDE + " feet.");
    }

    @Override
    public void swim() {
        System.out.println("The duck swims down to " + MAX_DEPTH + " feet.");
    }
}

// Main class to run the example
public class InterfaceExample {
    public static void main(String[] args) {
        Duck d = new Duck();
        d.fly();
        d.swim();
    }
}
```


Lab 7-6
Types


Java™