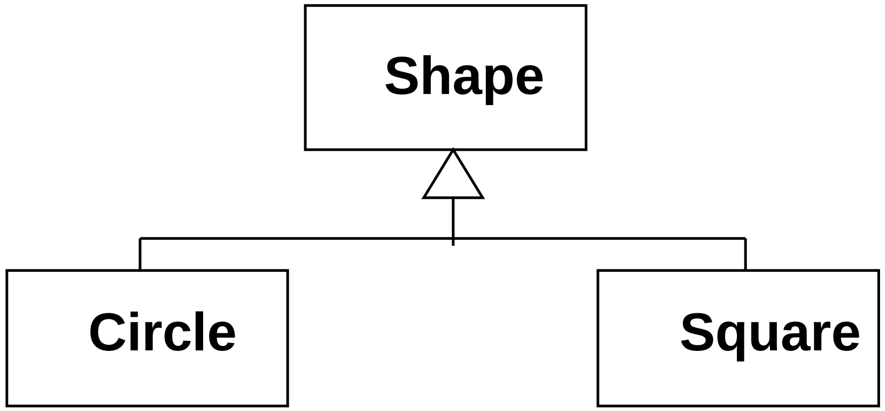
Dynamic binding

- Are the following legal (or compilable), given the class hierarchy?
  - object variables can refer to objects or their declared type AND any objects that are subclasses of the declared type

```
Shape s1, s2;
s1 = new Square();
s2 = new Circle();
```



Dynamic binding

- Consider the following class declarations:

```
public class BoardSpace

public class Property extends BoardSpace

public class Street extends Property

public class Railroad extends Property
```

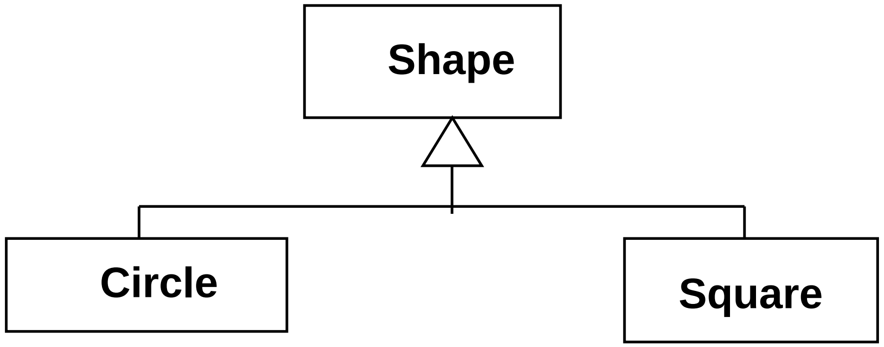
- Which of the following statements would cause a syntax (compilation) error? Assume all classes have a default constructor.

  A. `Object obj = new Railroad();`
  B. `Street s = new BoardSpace();`
  C. `BoardSpace b = new Street();`
  D. `Railroad r = new Street();`

Dynamic binding

- object variables have:
  - ► a *declared type:* also called the static type.
  - ► a *dynamic type:* the actual type at run time or when a particular statement is executed
    - This is also referred as dynamic binding

```
Shape s1; // declared type
s1 = new Square(); // dynamic type

Shape s2; // s2 is declared as 'Shape'
s2 = new Circle(); // but it is actually a 'Circle'
```



Abstract class

- Now, a question here

```
public class Shape
{
   String name;
   double area;

   // calculate the area
   public double calculateArea()
   {
      … … // how to calculate the area of a shape?
   }
}
```

- Is the word or definition of the ‘shape’ too general or abstract to compute its area? How about ‘Circle’ or ‘Square’?

Abstract class

- A class defined with the keyword ‘abstract’ is called abstract class
  - What about the method ‘calculateArea()’?

```
public abstract class Shape
{
   String name;
   double area;

   // calculate the area
   public double calculateArea()
   {
     … … // how to calculate the area of a shape?
   }
```

Abstract class

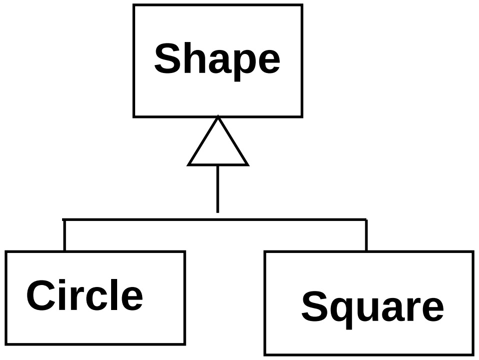
- A class defined with the keyword ‘abstract’ is called abstract class
  - ► What about the method ‘calculateArea()’?

```
public abstract class Shape
{
   String name;
   double area;

~~   // calculate the area~~
~~   public double calculateArea()~~
~~   {~~
~~       … … // how to calculate the area of a shape?~~
~~   }~~

   public abstract double calculateArea();
}
```

Abstract class

- How about the classes ‘Circle’ and ‘Square’ then?

```
public abstract class Shape
{
  String name;
  double area;
  public abstract double calculateArea();
}
```



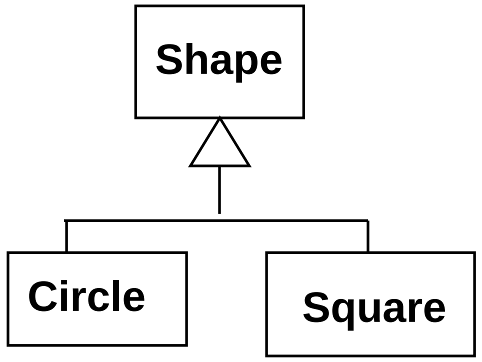
```
public class Circle extends Shape
{
  double radius;
  public double calculateArea(){
    area = radius * radius;
    return area;
  }
}
```

- This is method **overriding**!

Abstract class

- How about the classes ‘Circle’ and ‘Square’ then?

```
public abstract class Shape
{
  String name;
  double area;
  public abstract double calculateArea();
}
```



```
public class Square extends Shape
{
  double side;
  public double calculateArea(){
    area = side * side;
    return area;
  }
}
```

Abstract class

- What happens to the following?

```
public abstract class Shape {
  … …
  public abstract double calculateArea();
}
```

```
public class Square extends Shape {
  double side;
  public double calculateArea(){
    area = side * side;
    return area;
  }
}
```

```
public class Circle extends Shape
  double radius;
  public double calculateArea(){
    area = 3.14 * radius * radius;
    return area;
  }
}
```

- Which method to call?

```
public static void
main()
{
   Shape s1, s2;

   s1 = new Square();
   s1.side = 2.5;
   s1.area =
s1.calculateArea();

   s2 = new Circle();
   s2.radius = 3;
   s2.area =
s2.calculateArea();
```

Abstract class

- What happens to the following?

```
public abstract class Shape {
  … …
  public abstract double calculateArea();
}
```

```
public class Square extends Shape {
  double side;
  public double calculateArea(){
    area = side * side;
    return area;
  }
}
```

```
public class Circle extends Shape
  double radius;
  public double calculateArea(){
    area = 3.14 * radius * radius;
    return area;
  }
}
```

- Which method to call?

```
public static void
main()
{
  Shape s1, s2;

  s1 = new Square();
  s1.side = 2.5;
  s1.area =
s1.calculateArea();

  s2 = new Circle();
  s2.radius = 3;
  s2.area =
s2.calculateArea();
```

- This is also **dynamic binding**!

Abstract class

- One more example here

```
Abstract class Bird {
   public abstract void sound();
 }

class Crow extends Bird {
    public void sound() { System.out.println("caw"); }
 }

class Pigeon extends Bird {
    public void sound() { System.out.println("coo"); }
 }
```

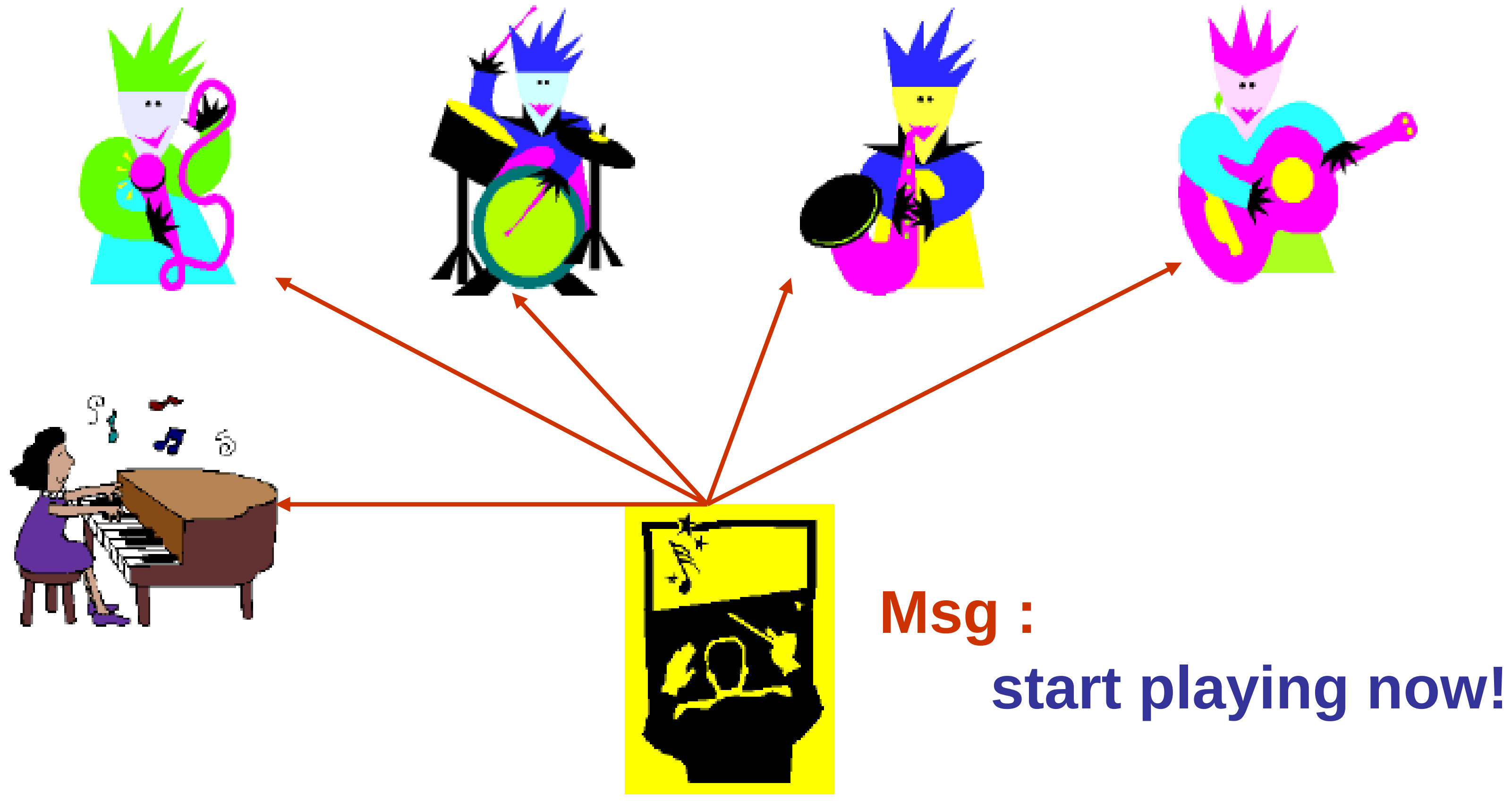
```
public static void main(String args[])
{
  Bird c, p;
  c = new Crow();
  p = new Pigeon();

  c.sound();
  p.sound()
}
```

Most important features of OO programming

- encapsulation
- inheritance
- **polymorphism**

Polymorphism

- Ability of different objects to perform the appropriate methods in response to the same message



Polymorphism

- Let’s have a demo
  - #1: class musician
  - #2: class shape

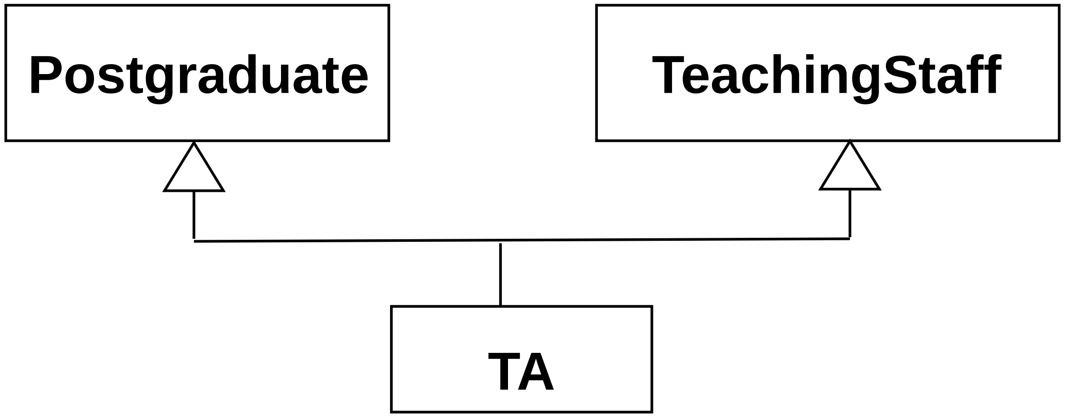
Polymorphism - motivation

- Inheritance allows code reuse, so programs are completed faster (especially large programs)

- Polymorphism allows code reuse in another way
  - Especially when an algorithm is essentially the same, but the code would vary based on the data type
    - This is a kind of genericity. Some languages support it via *templates, e.g., C++.*

Multiple Inheritance

- The are classes where the “is-a” test is true for more than one other class
  - ► a teaching assistant is a POSTGRADUATE
  - ► but he/she is also a TEACHING_STAFF



Multiple Inheritance

- So, basically it is a multiple inheritance.

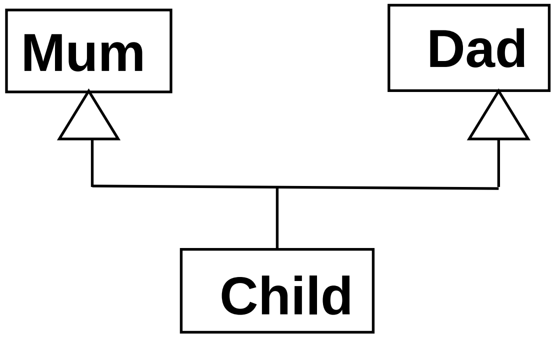
- Java requires all classes to inherit from one and only one other class, i.e.
  - some object-oriented languages do, such as C++ and Python

  - **Question**: Why Java does not allow it?

Multiple Inheritance

- Why Java NOT support more than one super/parent class:
  - Suppose both parents have defined a method, e.g., setName(String s), which method will this class invoke:

```
Child ch = new Child();
ch.setName("Charlie Dickens");
```



  - Remember the chain of construction?
    - Both parents have a default constructor, which one will be invoked by below?

```
Child ch = new Child();
```

Multiple inheritance

- How to provide a similar concept/idea of multiple inheritance while avoiding the possibility of conflicting implementations at the same time?

  - ► **Use Java ‘interface’!**

Interface in Java

- Allow for multiple, different implementations.
- Provides a way of creating *abstractions.*
  - a central idea of computer science and programming.
  - specify "what" without specifying "how"
  - "Abstraction is a mechanism and practice to reduce and factor out details so that one can focus on a few concepts at a time. "

```
public interface NameOfInterface
{
       public void method_1();
       public void method_2();
}
```

Interface in Java

- All methods in interfaces are public and abstract

```
public interface Mammal
{
       public void giveBirth();
       public void produceMilk();
}
```

- No constructors
- No instance variables

```
public interface Mammal
{
       ~~String name;~~
       ~~Mammal();~~
       public void giveBirth();
       public void produceMilk();
}
```

Interface in Java

- How to use the interface?

```
public interface Mammal
{
        public void giveBirth();
        public void produceMilk();
}
```

  - ► By implementation

```
Public class Whale implements Mammal
{
  public void giveBirth() { … … // how whale gives birth }
  public void produceMilk(); { … // how whale produces milk }
}
```

Interface in Java

- How to use the interface?
  - A class that implements an interface must provide implementations of **ALL methods** declared in the interface

```
public interface Mammal
{
        public void giveBirth();
        public void produceMilk();
}
```

```
Public class Whale implements Mammal
{
  public void giveBirth() { … … // how whale gives birth }
  public void produceMilk(); { … // how whale produces milk }
}
```

Why interface

- Allow the creation of *abstract data types*
  - "A set of operations that are precisely specified independent of any particular implementation. "
- Allow a class to be specified without worrying about the implementation
  - do design first
  - Don’t worry about implementation until design is done.

```
public interface Mammal
{
        // A special type of mammal could implement
these methods differently, e.g., Rabbit, Tiger.

        public void giveBirth();
        public void produceMilk();
}
```

Interface in Java

- A class inherits (extends) exactly one other class, but it can *implement* as many interfaces as it likes
  - allowing a form of multiple inheritance.

```
public interface Mammal
{
       public void giveBirth();
       public void produceMilk();
}
```

Interface in Java

- A class implementing more than one interface

```
public interface Mammal {
        public void giveBirth();
        public void produceMilk();
}
```

```
public interface MarineAnimal {
        public void swim();
}
```

```
public class Whale imlments Mammal, MarineAnimal
{
        public void giveBirth() { … … }
        public void produceMilk() { … … }
        public void swim() { … … }
}
```

Implementing an interface

- Let’s have a demo

Abstract class vs interface

Abstract class vs. interface

```
Abstract class Bird {
   public abstract void
sound();
 };

class Crow extends public Bird {
    public void sound() {
        cout << "caw" << endl; }
 };

class Pigeon extends public Bird {
    public void sound() {
        cout << "coo" << endl; }
 }
```

```
interface Bird {
   public void sound();
 };

class Crow implements Bird {
    public void sound() {
        cout << "caw" << endl; }
 };

class Pigeon implements Bird {
    public void sound() {
        cout << "coo" << endl; }
 }
```

- Similar? Differences between the two?

Abstract class vs Interface

- Common features
  - Neither can be instantiated

```
Abstract class BirdAbs {
    public abstract void
sound();
  }
```

```
interface BirdInt {
    public void sound();
  }
```

```
Public static void main() {
~~BirdInt bi = new BirdInt();~~
~~BirdAbs ba = new BirdAbs();~~

    BirdAbs ba; BirdInt bi;
}
```

  - Both need to implement those abstract methods

```
class Crow implements BirdInt {
    public void sound() {
        cout << "caw" << endl; }
 }
```

```
class Crow extends public BirdAbs
{
    public void sound() {
        cout << "caw" << endl; }
 }
```

Abstract class vs Interface

- Different features
  - 1. A class can extend only one abstract class, but may implement more than one interface

```
Abstract class BirdAbs_1 {
    public abstract void
sound();
 }
```

```
Abstract class BirdAbs_2 {
    public abstract void
eat();
 }
```

```
class Crow implements BirdInt_1,
         BirdInt_2
{
     public void sound() {
         cout << “caw” << endl; }
     public void eat() { … }
 }
```

```
interface BirdInt_1 {
    public void sound();
 }
```

```
interface BirdInt_2 {
    public void eat();
 }
```

```
class Crow extends BirdAbs_1,
BirdAbs_2 {
      ….
 }
```

Abstract class vs Interface

- Different features
  - Any problem with the followings?

```
interface BirdInt_1 {
    public void sound();
 }
```

```
interface BirdInt_2 {
    public void sound();
 }
```

```
class Crow implements BirdInt_1,
      BirdInt_2
{
    public void sound() {
      cout << "caw" << endl; }
 }
```

```
class Crow extends BirdAbs_1
implements BirdInt_1 {
   public void eat() { … }
   public void sound { … }
 }
```

```
Abstract class BirdAbs {
    public abstract void
eat();
 }
```

Abstract class vs Interface

- Different features
  - 2. An abstract class can have methods already implemented, but this is not for an interface
    - New Java version supports this by having a default method
      - https://www.programiz.com/java-programming/interfaces
  - Note that all methods in interface are public, even you do not declare them.

```
Abstract class BirdAbs_1 {
   public abstract void
eat();

    public void fly { … }
 }
```

```
interface BirdInt_1 {
   public void sound();

    public void perch ()
      { … … ? }
 }
```

```
interface BirdInt_1 {
  void sound();

 }
```

Abstract class vs Interface

- Different features
  - Any problem with the followings?

```
Abstract class BirdAbs {
    public abstract void eat();

    public void perch ()
    { … … }
 }
```

```
class Crow extends BirdAbs
 {
   public void perch() { … }
 }
```

Abstract class vs Interface

- Apart from obvious reasons such as gaining the feature of multiple inheritance, when to use abstract class, and when to use interface?

  - When a class ‘extend’ an abstract class, it is more (closely) related to the abstract class, e.g., Rectangle vs. Shape, Crow vs. Bird.
    - ‘is-a’ relationship normally stands!
    - So, “class Whale imlments Mammal, MarineAnimal” is not a good idea of design?

Abstract class vs Interface

- Apart from obvious reasons such as gaining the feature of multiple inheritance, when to use abstract class, and when to use interface?
  - When a class ‘implements’ an interface, it could be very loosely related the interface.
    - The java interface, Comparable, can be implemented by classes that are quite different, e.g., ‘Bird’, ,’Rectangle’, or anything that can be compared!
    - When it is not concerned about who implements the behavior, but just providing a contract (interface)

```
class Image implements Sender
{
   void send (“a file to send”) { … … }
 }
```

```
public interface Sender
{
   void send (“a file to send”);
 }
```

```
class Video implements Sender
{
   void send (“a file to send”) { … … }
 }
```

Abstract class vs Interface

- The differences between abstract class and interface, and in what scenario we should use them, is a very interesting and yest intrigue question!

  - Other implications too such as code maintenance

  - Read more here, https://stackoverflow.com/questions/10040069/abstract-class-vs-interface-in-java, you’re interested in digging deep!