Inheritance and Private

- **Private members**:
  - Child class **inherits all the private data** of Base class
  - However, **private members** of the base class **cannot** be accessed directly

- **Why is this?** After you have gone to all the work of setting up privacy, it wouldn't be fair to allow someone to simply **extend** your class and now have access to all the **private** information

Quiz 5: True/False

Excepting Object, which has no superclass, every class has one and only one direct superclass.

A. True
B. False

Quiz5: True/False

Excepting Object, which has no superclass, every class has one and only one direct superclass.

A. True
B. False

Quiz 6:

```
class Base {
  public void foo(){
    println("Base");
  }
}
class Derived extends Base {
  private void foo(){
    println("Derived");
  }
}
…
  Base b = new Derived();
  b.foo();
…
```

A. Base
B. Derived
C. Compiler Error
D. Runtime Error

Quiz 6:

```
class Base {
  public void foo(){
    println("Base");
  }
}
class Derived extends Base {
  private void foo(){
    println("Derived");
  }
}
…
  Base b = new Derived();
  b.foo();
…
```

A. Base
B. Derived
C. Compiler Error
D. Runtime Error

It is compiler error to give more restrictive access to a derived class function which overrides a base class function.

Quiz 7:

class Animal has a subclass Mammal. Which of the following is true:

A. Because of single inheritance, Mammal can have no subclasses.
B. Because of single inheritance, Mammal can have no other parent than Animal.
C. Because of single inheritance, Animal can have only one subclass.
D. Because of single inheritance, Mammal can have no siblings.

Quiz 7:

class Animal has a subclass Mammal. Which of the following is true:

A. Because of single inheritance, Mammal can have no subclasses.
B. Because of single inheritance, Mammal can have no other parent than Animal.
C. Because of single inheritance, Animal can have only one subclass.
D. Because of single inheritance, Mammal can have no siblings.

Access level

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Object

- Object is the superclass of all java classes

- The class **Object** has no instance variables, but defines a number of methods. These include:

  toString( )*: returns a String representation of this object*

  equals(Object o): test for equality with another object o

- Every class you define should, overrides these two methods with something that makes sense for your class (hashCode method is also included in the group)

Early and Late Binding

- **Motivation**: Consider the following example:

```
Base b = new Child();
b.toString();
```

- **Q**: Should this call **Base's** toString or **Child's** toString?
- **A**: There are good arguments for either choice:

  Early (static) binding: The variable b is **declared** to be of type **Base**. Therefore, we should call the Base's toString

  Late (dynamic) binding: The object to which b refers was **created** as a "new **Child**". Therefore, we should call the Child's toString

  Pros and cons: Early binding is more efficient, since the decision can be made at compile time. Late binding provides more flexibility

- **Java uses late binding** (by default): so Faculty toString is called
  (**Note**: C++ uses early binding by default.)

Polymorphism

- Java's **late binding** makes it possible for a single reference variable to refer to objects of many different types. Such a variable is said to be **polymorphic** (meaning having many forms)
- **Example**: Create an array of various university people and print

```
Shape[ ] list = new Shape[3];
list[0] = new Rect(10,20);
list[1] = new Circle (10);
list[2] = new Triangle(3,4,5)
for (int i = 0; i < list.length; i++ )
     System.out.println( list[i].getArea( ) );
```

Output:

- **What type is list[i]?** It can be a reference to any object that is derived from **Shape**. The appropriate **getArea** will be called

getClass and instanceof

- Objects in Java can access their type information **dynamically**
- **getClass( )**: Returns a representation of the class of any object

```
Person bob = new Person( … );
Person ted = new Student( … );

if ( bob.getClass( ) == ted.getClass( ) ) // false (ted
  is really a Student)
```

- **instanceof**: You can determine whether one object is an instance of (e.g., derived from) some class using **instanceof**. Note that it is an **operator** (!) in Java, not a method call

Up-casting and Down-casting

- We have already seen that we can assign a derived class reference anywhere that a base class is expected

  Upcasting: Casting a reference **to a base class** (casting up the inheritance tree). This is done **automatically** and is **always safe**

  Downcasting: Casting a reference **to a derived class**. This may **not be legal** (depending on the actual object type). You can **force** it by performing an explicit cast

- Illegal downcasting results in a **ClassCastException** run-time error

Safe Downcasting

- Can we check for the **legality** of a cast before trying it?
- **A**: Yes, using **instanceof**.

```
For(s:Shape){
        if(s instanceof Circle){
                Circle c = s;
                int r = c.getRadius();
        }
}
```

Only Circle has getRadius method

Disabling Overriding with “final”

- Sometimes you do not want to allow method overriding

  Correctness: Your method only makes sense when applied to the base class. Redefining it for a derived class might break things

  Efficiency: Late binding is less efficient than early binding. You know that no subclass will redefine your method. You can force early binding by disabling overriding

- We can disable overriding by declaring a method to be “**final**”

Disabling Overriding with “final”

- **final**: Has two meanings, depending on context:
  - Define **symbolic constants**:

```
public static final int MAX_BUFFER_SIZE = 1000;
```

  - Indicate that a method **cannot be overridden by derived classes**

```
public class Parent {
   …
   public final void someMethod( ) { … }
}

public class Child extends Parent {
   …
   public void someMethod( ) { … }
}
```

```
Subclasses cannot
override this method
```

```
Illegal! someMethod is
final in base class.
```

Quiz 8

```
class Base {
  final public void show() {
    println("Base");
  }
}
class Derived extends Base {
  public void show() {
    println("Derived");
  }
}
class Main {
  public static void(String[] args){
    Base b = new Derived();
    b.show();
   }
}
```

A. Base
B. Derived
C. Compiler Error
D. Runtime Error

Quiz 8

```
class Base {
  final public void show() {
    println("Base");
  }
}
class Derived extends Base {
  public void show() {
    println("Derived");
  }
}


…
    Base b = new Derived();
    b.show();
…
```

A. Base
B. Derived
C. Compiler Error
D. Runtime Error

Final methods cannot be overridden. Compiler Error: overridden method is final

Quiz 9

```
class Base {
  public static void show() {
    println("Base”);
  }
}
class Derived extends Base {
  public static void show() {
    println("Derived");
  }
}
…
  Base b = new Derived();;
  b.show();
…
```

A. Base
B. Derived
C. Compiler Error

Quiz 9

```
class Base {
  public static void show() {
    println("Base");
  }
}
class Derived extends Base {
  public static void show() {
    println("Derived");
  }
}
…
  Base b = new Derived();;
  b.show();
…
```

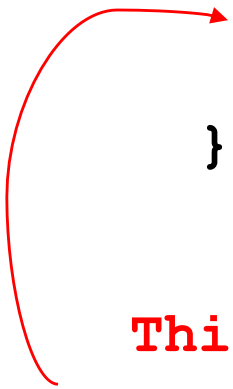
A. Base
B. Derived
C. Compiler Error

when a function is static, runtime polymorphism doesn't happen.

Abstract Class

- Abstract classes cannot be instantiated, but they can be subclassed.
- It may or may not include abstract methods.

```
public abstract class Shape {
  private String id;
  public Shape (String id) {this.id = id};
  public abstract double getArea();
  public String getId() {return id;}
}
```



This abstract method must be defined in a concrete subclass.

Abstract Class

```
public abstract class Shape {
  private String id;
  public Shape (String id) {this.id = id};
  public abstract double getArea();
  public String getId() {return id;}
}

public class Circle extends Shape {
  private double radius;
  public Circle (double r) {
    super("Circle"); radius = r;
  }
  double getArea(){return Math.PI * radius * radius;}
  public double getRadius() {return radius;}
  public void setRadius(double r) {radius = r}
}
```

Must implement