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:
  
  ```java
  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

  ```java
  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

```java
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`.

```java
For(s:Shape){
    if(s instanceof Circle){
        Circle c = (Circle)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*:
    ```java
    public static final int MAX_BUFFER_SIZE = 1000;
    ```
  - Indicate that a method cannot be overridden by derived classes
    ```java
    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.
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();
    }
}
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();
...
Quiz 9

```java
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

doesn't happen.
Abstract Class

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

```java
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.
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}
}
Inheritance versus Composition

• **Inheritance** is but one way to create a complex class from another. The other way is to explicitly have an instance variable of the given object type. This is called **composition**.

```java
Common Object:
public class ObjA {
    public methodA() { ... }
}
```

```java
Inheritance:
public class ObjB extends ObjA {
    ObjB {
        ... // call methodA();
    }
}
```

```java
Composition:
public class ObjB {
    ObjA a;
    // call a.methodA();
}
```

• **When should I use inheritance vs. Composition?**
  • **ObjB “is a” ObjA**: in this case use **inheritance**
  • **ObjB “has a” ObjA**: in this case use **composition**

Add ObjA as an instance variable.
Inheritance versus Composition

- **University parking lot permits**: A parking permit object involves a university Person and a lot name ("4", "11", "XX", "Home Depot")

  **Inheritance**:  
  ```java
  public class Permit extends Person {
    String lotName;
    // ...
  }
  ```

  **Composition**:  
  ```java
  public class Permit {
    Person p;
    String lotName;
    // ...
  }
  ```

- **Which to use?**
  - A parking permit “is a” person? Clearly no
  - A parking permit “has a” person? Yes, because a Person is one of the two entities in a permit object
  - So composition is the better design choice here

- **Prefer Composition over inheritance**
  When in doubt or when multiple choices available, prefer composition over inheritance