Object Oriented Programming

1. Inheritance and polymorphism
2. Classes Object and Class

How to Reuse Code?

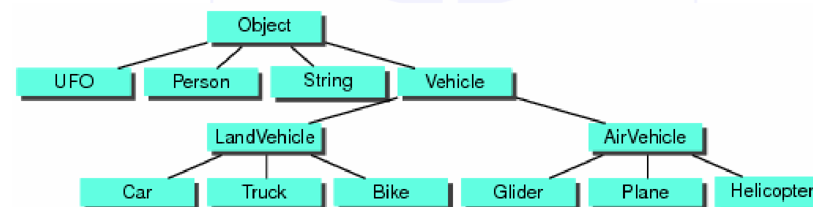
- Write the class completely from scratch (one extreme).
  - What some programmers always want to do!
- Find an existing class that exactly match your requirements (another extreme).
  - The easiest for the programmer!
- Built it from well-tested, well-documented existing classes.
  - A very typical reuse, called composition reuse!
- Reuse an existing class with inheritance
  - Requires more knowledge than composition reuse.

Introduction to Inheritance

- *Inheritance* is one of the main techniques of object-oriented programming
- Using this technique:
  - a very general form of a class is first defined and compiled, and then
  - more specialized versions of the class are defined by adding instance variables and methods
- The specialized classes are said to *inherit* the methods and instance variables of the general class

Introduction to Inheritance

- Inheritance models "*is a*" relationships
  - Object "is an" other object if it can behave in the same way
  - Inheritance uses *similarities and differences* to model groups of related objects
- Where there's Inheritance, there's an *Inheritance Hierarchy* of classes



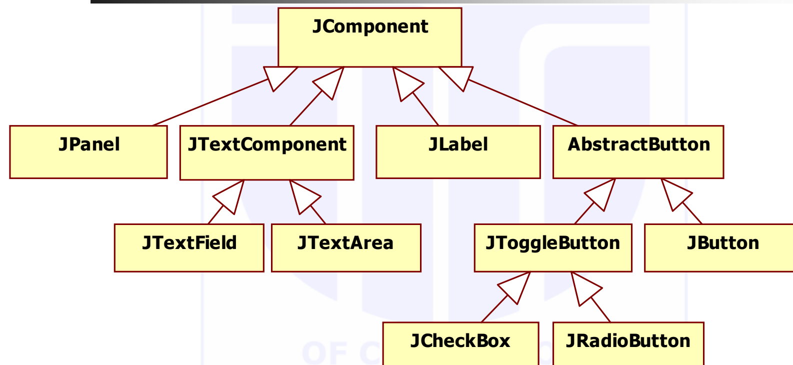
Introduction to Inheritance

- Inheritance is a way of:
  - organizing information
  - grouping similar classes
  - modeling similarities between classes
  - creating a taxonomy of objects
- `Vehicle` is called *superclass*
  - or *base class* or *parent class*
- `LandVehicle` is called *subclass*
  - or *derived class* or *child class*
- Any class can be both at same time
  - E.g., `LandVehicle` is superclass of `Truck` and subclass of `Vehicle`

Introduction to Inheritance

- Can only inherit from *one* superclass in Java
  - otherwise, no limit to depth or breadth of class hierarchy
  - C++ allows a subclass to inherit from multiple superclasses (error-prone)
- Note that, in Java, every class extends the `Object` class either directly or indirectly
- A derived class automatically has all the instance variables and methods that the base class has, and it can have additional methods and/or instance variables as well
- Inheritance is especially advantageous because it allows code to be *reused*, without having to copy it into the definitions of the derived classes

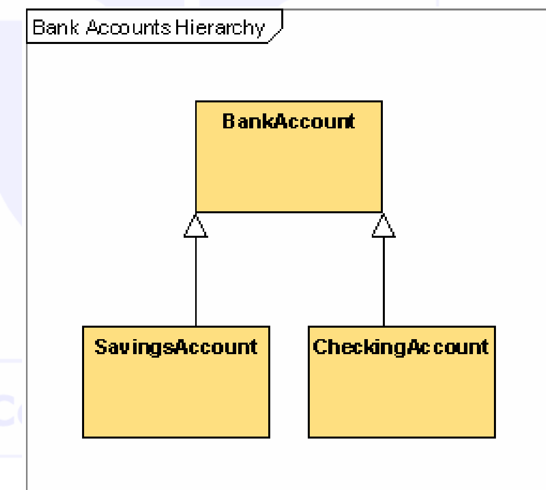
Example. A Part of the Hierarchy of Swing User Interface Components



- Superclass `JComponent` has methods `getWidth`, `getHeight`
- `AbstractButton` class has methods to set/get button text and icon

A Simpler Hierarchy: Hierarchy of Bank Accounts

- **Inheritance hierarchy:**



A Simpler Hierarchy: Hierarchy of Bank Accounts

- Brief specification
  - All bank accounts support the `getBalance` method
  - All bank accounts support the `deposit` and `withdraw` methods, but the implementations differ
  - Checking account needs a method `deductFees`; savings account needs a method `addInterest`

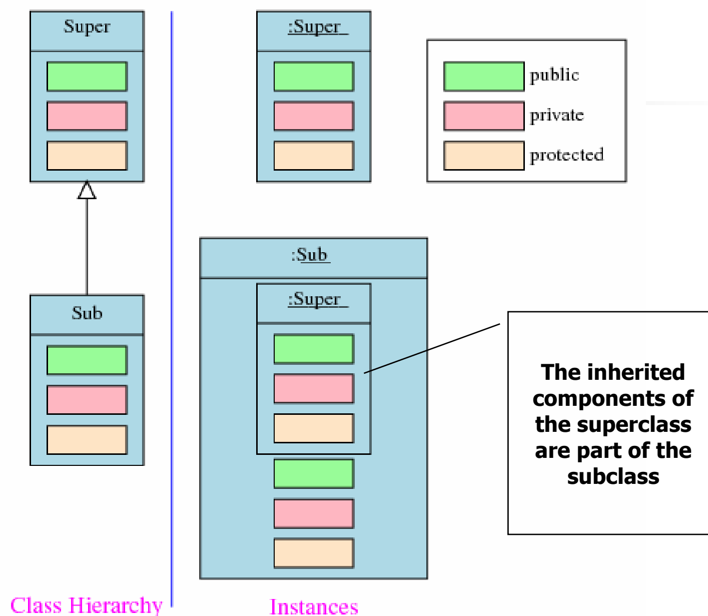
Derived Classes

- Since an savings account is a bank account, it is defined as a *derived* class of the class `BankAccount`
  - A *derived class* is defined by adding instance variables and/or methods to an existing class
  - The phrase `extends BaseClass` must be added to the derived class definition:

```
public class SavingsAccount extends BankAccount
```

- Syntax for inheritance:

```
class SubclassName extends SuperclassName
{
    methods
    instance fields
}
```



An Example: Employees

- See with BlueJ

Derived Classes

- When a derived class is defined, it is said to inherit the instance variables and methods of the base class that it extends
  - Class **`Employee`** defines the instance variables **`name`** and **`hireDate`** in its class definition
  - Class **`HourlyEmployee`** also has these instance variables, but they are not specified in its class definition
  - Class **`HourlyEmployee`** has additional instance variables **`wageRate`** and **`hours`** that are specified in its class definition

Derived Classes

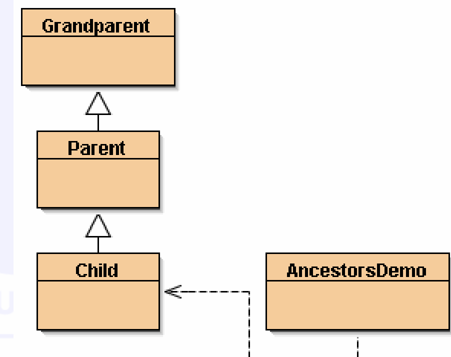
- Just as it inherits the instance variables of the class **`Employee`**, the class **`HourlyEmployee`** inherits all of its methods as well
  - The class **`HourlyEmployee`** inherits the methods **`getName`**, **`getHireDate`**, **`setName`**, and **`setHireDate`** from the class **`Employee`**
  - Any object of the class **`HourlyEmployee`** can invoke one of these methods, just like any other method

Derived Class (Subclass)

- A derived class, also called a *subclass*, is defined by starting with another already defined class, called a *base class* or *superclass*, and adding (and/or changing) methods, instance variables, and static variables
  - The derived class *inherits* all the *public methods*, all the *public and private instance variables*, and all the *public and private static variables* from the base class
  - The derived class *can add more* instance variables, static variables, and/or methods
- *Definitions* for the inherited variables and methods *do not appear* in the derived class
  - The code is reused without having to explicitly copy it, unless the creator of the derived class *redefines* one or more of the base class *methods*

Parent and Child Classes

- A base class is often called the *parent class*
  - A derived class is then called a *child class*
- These relationships are often extended such that a class that is a parent of a parent . . . of another class is called an *ancestor class*
  - If class **`A`** is an ancestor of class **`B`**, then class **`B`** can be called a *descendent* of class **`A`**



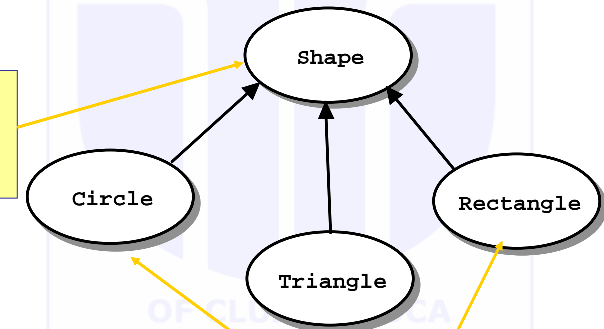
Abstract Classes

- An abstract method or class is declared with the `abstract` keyword. E.g.

```
public abstract double calcPay(
double hours );
```

- No object can be instantiated from an abstract class
- Every subclass of an abstract class that will be used to instantiate objects must provide implementations for all abstract methods in the superclass.
- Abstract classes save time, since we do not have to write "useless" code that would never be executed.
- An abstract class can inherit *abstract* methods
  - From an interface, or
  - From a class.

Example: A `Shape` Class



Superclass: contains abstract methods `area` and `perimeter`.

Subclasses: implement concrete methods `area` and `perimeter`.

Example: A `Shape` Class

```
public abstract class Shape {
    private static int counter;
    // Constructor
    public Shape()      {
        counter++;
    }
    // calculate area
    public abstract double area();
    // calculate perimeter
    public abstract double perimeter();
    // get number of shapes
    public int getCount() {
        return counter;
    }
    protected void finalize() throws Throwable {
        counter--;
    }
}
```

Superclass definition. Note class is declared `abstract`.

Abstract method definitions. Note only header is declared. These methods **must be overridden** in all concrete subclasses.

Example: the `Circle` Subclass

```
public class Circle extends Shape {
    private double r; // radius of circle
    // Constructor
    public Circle(double r)     {
        super();
        this.r = r;
    }
    // calculate area
    public double area()    {
        return  Math.PI * r * r;
    }
    // calculate perimeter
    public double perimeter()    {
        return 2.0 * Math.PI * r ;
    }
     protected void finalize() throws Throwable    {
        super.finalize();
    }
}
```

Concrete class. Class *must not* contain or inherit abstract methods. Inherited abstract methods must be overridden.

Concrete method definitions. Note that the method body is declared here.

Example: the `Triangle` Subclass

```
public class Triangle extends Shape {
    private double s; // side of Triangle
    // Constructor
    public Triangle(double s)     {
        super();
        this.s = s;
    }
    // calculate area
    public double area()     {
        return ( Math.sqrt(3.)/4 * s *s );
    }
    // calculate perimeter
    public double perimeter()     {
        return 3.0 * s ;
    }
     protected void finalize() throws Throwable {
        super.finalize();
    }
}
```

Concrete class. Class *must not* contain or inherit abstract methods. Inherited abstract methods must be overridden.

Concrete method definitions. Note that the method bodies are different from those in `Circle`, but the method signatures are *identical*.

Other subclasses of **`Shape`** will also override the abstract methods **`area`** and **`perimeter`**

Example: Class `TestShape`

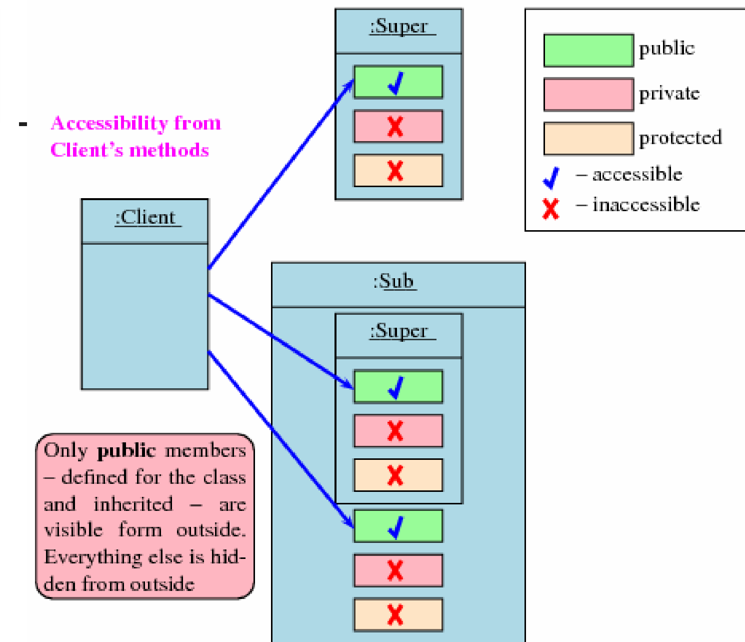
```
public class TestShape
{
    public static void main(String[] args)
    {
        // Create an array of Shapes
        Shape s[] = new Shape[2];
        // create objects
        s[0] = new Circle(2);
        s[1]= new Triangle(2);
        // Print out the number of Shapes
        System.out.println(s[0].getCount() + " shapes created");
        for (int i = 0; i < s.length; i++ ) {
            System.out.print(s[i].toString() + " ");
            System.out.print("Area = " + s[i].area());
            System.out.println(" Perimeter = " + s[i].perimeter());
        }
    }
}
```

Create objects of subclasses using superclass references.

Call area and perimeter methods. The proper version of each method will be automatically called for each object.

`protected` Instance Variables

- As a general pattern, subclasses:
  - Inherit `public` capabilities (methods)
  - Inherit `private` properties (instance variables) but do not have access
  - Inherit `protected` variables and can access them
- A variable that is declared `protected` by a superclass becomes *part of the inheritance*
  - variable becomes available for subclasses to access *as if it were their own*
  - in contrast, if an instance variable is declared `private` by a superclass, its subclasses won't have access to it
    - superclass could still provide protected access to its private instance variables via *accessor* and *mutator* methods

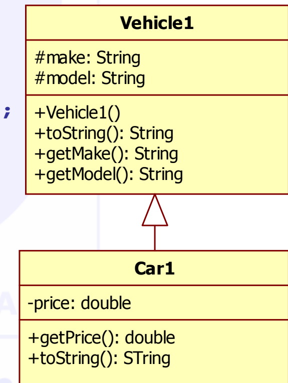


`protected` vs `private` Instance Variables

- How can I decide between `private` and `protected`?
  - use `private` if you want an instance variable to be *encapsulated* by superclass
    - e.g., doors, windows, spark plugs of a car
  - use `protected` if you want an instance variable to be available for subclasses to change (and you don't want to make the variable more generally accessible via accessor/mutator methods)
    - e.g., engine of a car

`protected`, Example

```
public class Vehicle1 {
  protected String make;
  protected String model;
  public Vehicle1() { make = ""; model = "";}
  public String toString() {
    return "Make: " + make + " Model: " + model;
  }
  public String getMake(){ return make;}
  public String getModel() { return model;}
}
public class Car1 extends Vehicle1 {
  private double price;
  public Car1() { price = 0.0; }
  public String toString() {
    return "Make: " + make + " Model: " + model
      + " Price: " + price;
  }
  public double getPrice(){ return price; }
}
```



Overriding a Method Definition

- Although a derived class inherits methods from the base class, it can change or *override* an inherited method if necessary
  - In order to override a method definition, a new definition of the method is simply placed in the class definition, just like any other method that is added to the derived class
- Ordinarily, the type returned may not be changed when overriding a method
- However, if it is a *class type*, then the returned type may be changed to that of any *descendent class* of the returned type
- This is known as a *covariant return type*
  - *Covariant return types* are new in Java 5.0; they are not allowed in earlier versions of Java

Covariant Return Type

- Given the following base class:

```
public class BaseClass
{ . . .
  public Employee getSomeone(int someKey)
  . . .
```

- The following is allowed in Java 5.0:

```
public class DerivedClass extends BaseClass
{ . . .
  public HourlyEmployee getSomeone(int someKey)
  . . .
```

Changing the Access Permission of an Overridden Method

- The access permission of an overridden method can be changed *from private* in the *base class* to *public* (or some other *more permissive access*) in the *derived class*
- However, the access permission of an overridden method *can not be changed* from public in the base class *to a more restricted access* permission in the derived class
  - I.e. we can relax access permission in a derived class

Changing the Access Permission of an Overridden Method

- Given the following method header in a base class:
  `private void doSomething()`
- The following method header is valid in a derived class:
  `public void doSomething()`
- However, the opposite is not valid
- Given the following method header in a base class:
  `public void doSomething()`
- The following method header is not valid in a derived class:
  `private void doSomething()`

Pitfall: Overriding Versus Overloading

- Do not confuse *overriding* a method in a derived class with *overloading* a method name
  - When a method is *overridden*, the new method definition given in the derived class has the *exact same number and types of parameters as in the base class*
  - When a method in a derived class has a *different signature* from the method in the base class, that is *overloading*
  - Note that when the *derived class overrides* the original method, *it still inherits the original method* from the base class as well

The `final` Modifier

- If the modifier `final` is placed before the definition of a *method*, then that method *may not be redefined* in a derived class
- It the modifier `final` is placed before the definition of a *class*, then that *class may not be used as a base class* to derive other classes

The `super` Constructor

- A derived class uses a constructor from the base class to initialize all the data inherited from the base class
  - In order to invoke a constructor from the base class, it uses a special syntax:

```
public derivedClass(int p1, int p2, double p3)
{
  super(p1, p2);
  instanceVariable = p3;
}
```

  - In the above example, `super(p1, p2);` is a call to the base class constructor

The `super` Constructor

- A call to the base class constructor can never use the name of the base class, but uses the keyword `super` instead
- A call to `super` must always be the first action taken in a constructor definition
- An instance variable cannot be used as an argument to `super`
  - Why this is not allowed?

The `super` Constructor

- If a derived class constructor does not include an invocation of `super`, then the no-argument constructor of the base class will automatically be invoked
  - This can result in an error if the base class has not defined a no-argument constructor
- Since the inherited instance variables should be initialized, and the base class constructor is designed to do that, then *an explicit call to* `super` *should always be used*

Constructors and Subclasses

- The keyword `this` is used in a constructor to invoke another constructor of the same class. Example:

```
public class Circle extends ClosedFigure
{
  private int radius;
  public Circle(int radius)
  {
    super();
    this.radius = radius;
  }
  public Circle()
  {
    this(1); // invoke above constructor with argument 1
  }
…
}
```

Access to a Redefined Base Method

- Within the definition of a method of a derived class, the base class version of an overridden method of the base class can still be invoked
  - Simply preface the method name with super and a dot

```
public String toString()
{
  return (super.toString() + "$" + wageRate);
}
```

- However, using an object of the derived class outside of its class definition, there is no way to invoke the base class version of an overridden method

You Cannot Use Multiple `supers`

- It is only valid to use `super` to invoke a method from a direct parent
  - Repeating `super` will not invoke a method from some other ancestor class
- For example, if the `Employee` class were derived from the class `Person`, and the `HourlyEmployee` class were derived from the class `Employee` , it would not be possible to invoke the `toString` method of the `Person` class within a method of the `HourlyEmployee` class

```
super.super.toString() // ILLEGAL!
```

The `this` Constructor

- Within the definition of a constructor for a class, `this` can be used as a name for invoking another constructor in the same class
  - The same restrictions on how to use a call to `super` apply to the `this` constructor
- If it is necessary to include a call to both `super` and `this`, the call using `this` must be made first, and then the constructor that is called must call `super` as its first action

The `this` Constructor

- Often, a no-argument constructor uses `this` to invoke an explicit-value constructor
  - No-argument constructor (invokes explicit-value constructor using `this` and default arguments):

```
public ClassName()
{
  this(argument1, argument2);
}
```

  - Explicit-value constructor (receives default values):

```
public ClassName(type1 param1, type2 param2)
{
  . . .
}
```

The `this` Constructor

```
public HourlyEmployee()
{
  this("No name", new Date(), 0, 0);
}
```

- The above constructor will cause the constructor with the following heading to be invoked:

```
public HourlyEmployee(String theName, Date
  theDate, double theWageRate, double
  theHours)
```

Tip: An Object of a Derived Class Has More than One Type

- An object of a derived class has the type of the derived class, and it also has the type of the base class
- More generally, an object of a *derived class* has the *type of every one of its ancestor classes*
  - Therefore, an object of a derived class can be assigned to a variable of any ancestor type

Tip: An Object of a Derived Class Has More than One Type

- An object of a derived class can be plugged in as a parameter in place of any of its ancestor classes
- In fact, a derived class object can be used anyplace that an object of any of its ancestor types can be used
- Note, however, that this relationship does not go the other way
  - An ancestor type can never be used in place of one of its derived types

Pitfall: The Terms "Subclass" and "Superclass"

- The terms *subclass* and *superclass* are sometimes mistakenly reversed
  - A *superclass* or base class is *more general* and inclusive, but *less complex*
  - A *subclass* or derived class is *more specialized*, less inclusive, and *more complex*
    - As more instance variables and methods are added, the number of objects that can satisfy the class definition becomes more restricted

Abstract Class Use

- An abstract class factors out implementation of its concrete subclasses.
- Used to exploit polymorphism.
  - Functionality specified in parent class can be given implementations appropriate to each concrete subclass.
- Abstract class must be stable.
  - any change in an abstract class propagates to subclasses and their clients.
- A concrete class can only extend one (abstract) class

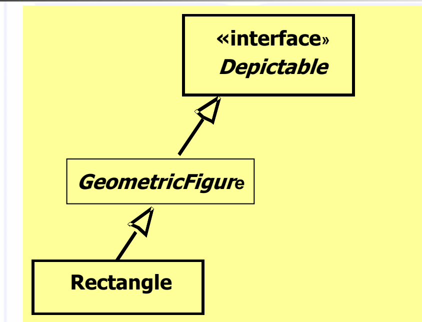
Interfaces, Abstract classes and Concrete Classes

- An ***interface***
  - used to specify functionality required by a client.
- An ***abstract class***
  - provides a basis on which to build concrete servers.
- A ***concrete class***
  - completes server implementation specified by an interface;
  - furnish run-time objects;
  - not generally suited to serve as a basis for extension.

Interface Use

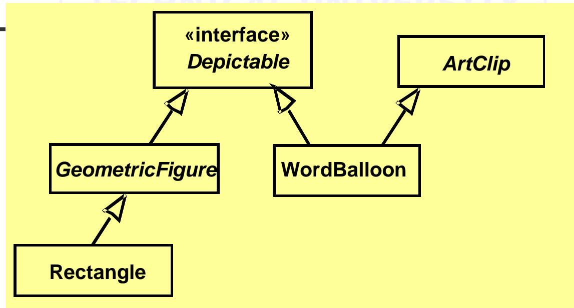
- Interfaces are by definition abstract.
  - separate an object's implementation from its specification.
  - they do not fix any aspect of an implementation.
- A class can implement more than one interface.
- Interfaces allow a more generalized use of polymorphism; instances of relatively unrelated classes can be treated as identical for some specific purpose.
- In your program designs, remember to use the Java *interface to share common behavior*. Use *inheritance to share common code*.

Interfaces, Abstract Classes, and Concrete Classes




```
public boolean isIn (Location point, Depictable figure)
{
        Location l = figure.getLocation();
        Dimension d = figure.getDimension();
        …
}
```

Interfaces, Abstract Classes, and Concrete Classes




```
public boolean isIn (Location point, Depictable figure)
{
        Location l = figure.getLocation();
        Dimension d = figure.getDimension();
        …
}
```

- Can pass instances of WordBalloon to `isIn`.

Polymorphism Example

```
public class Base {
  protected int theInt = 100;

  ...
  public void printTheInt() {
    System.out.println( theInt );
  }
```

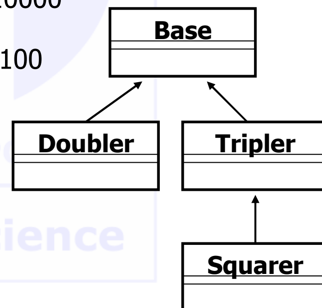
```
public class Doubler extends Base {
  ...
  public void printTheInt() {
    System.out.println( theInt*2 );
  }
```

```
public class Tripler extends Base {
  ...
  public void printTheInt() {
    System.out.println( theInt*3 );
  }
```

```
public class Squarer extends Tripler {
  ...
  public void printTheInt() {
          System.out.println( theInt*theInt );
  }
}
```

Polymorphism Example

```
Base theBase;
theBase = new Base();
theBase.printTheInt();          → 100
theBase = new Doubler();
theBase.printTheInt();          → 200
theBase = new Tripler();
theBase.printTheInt();          → 300
theBase = new Squarer();
theBase.printTheInt();          → 10000
theBase = new Base();
theBase.printTheInt();          → 100
…
```



subtype polymorphism

As dynamic binding occurs the behavior (i.e., methods) follow the objects.

Polymorphism

- A polymorphic variable can appear to change its type through dynamic binding.
- The compiler always understands a variable's type according to its declaration.
- The compiler permits some flexibility by way of type conformance.
- At run-time the behavior of a method call depends upon the type of the object and not the variable.
- Example:

```
Base theBase;
theBase = new Doubler();
theBase = new Squarer();
theBase.printTheInt();
```

Why is Polymorphism Useful?

- Polymorphism permits a superclass to capture commonality, leaving the specifics to subclasses.

```
public class AView {
  ...
  public double area() {
        return 0.0;
  }
```

Suppose that AView included an area method, as shown.

Then ARectangle should be written as ...

```
public class ARectangle extends AView {
  ...
  public double area() {
        return getWidth() * getHeight();
  }
}
```

and AOval should be written as ...

```
public class AOval extends AView {
  ...
  public double area() {
        return getWidth()/2. * getHeight()/2. * Math.PI;
  }
}
```

now consider

```
public double coverageCost( AView v, double costPerSqUnit) {
        return v.area() * costPerSqUnit;
}
```

Inheriting from `Object`

- *Every class in Java extends some other class*.
- If you don't explicitly specify the class that your new class extends, it will automatically extend the class named ***Object***.
- All classes in Java are in a class hierarchy where the class named *Object* is the root of the hierarchy.
- Some classes extend *Object* directly, while other classes are subclasses of *Object* further down the hierarchy.
- Class *Object* is defined in `java.lang`

Methods in the `Object` class

- *Object* defines default versions of the following methods:
  - `toString()` – returns a string (readable representation of the object)
  - `equals(Object obj)` – must be overridden for content equality
  - `hashCode()` – returns the hash-code value for the object; values are different for different objects
  - `getClass()` – returns an object of type Class; there is an object of type Class for each class of an application
  - `notify(), notifyAll(), wait(), wait(long timeout), wait(long timeout, int nanos)` – used with multithreading
  - `clone()` – creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object.
  - `finalize()` – intended to perform cleanup actions before the object is irrevocably discarded.

Equality

- There are two different kinds of equality:
  - *Identity equality* means that two expressions have the same identity. (I.e., they represent the same object.)
  - *Content equality* means that two expressions represent objects with the same value/content .

The == symbol tests for identity equality, when applied to reference data.

- Example:

```
AOval ov1, ov2;
ov1 = new AOval(0, 0, 100, 100);
ov2 = new AOval(0, 0, 100, 100);
if (ov1 == ov2){ System.out.println("identity equality");}
 else {System.out.println("content equality");}
```

- *The equals* method from `Object` can be used to provide a content equality check.

The class `Class`

- The class Class is defined as

```
public final class Class extends Object
    implements Serializable, …
```

- Instances of the class `Class` represent classes and interfaces in a running Java application.
- An object of type `Class` contains information about the class whose instance is the calling object
- It does not have a self constructor
- `Class` objects are constructed at run-time by the JVM
- Two ways of constructing objects of this type:
  - `getClass()` from class `Object`
  - `forName()` from class `Class` (static method)

The class `Class`

- Methods:
  - `public String getName()`
    - returns a String representing the name of the entity represented by the `Class` object `this`
    - the entity can be: class, interface, array, primitive type, void
  - `public static Class forName(String className) throws ClassNotFoundException`
    - returns an object of type `Class` that contains info about the class of the call object
  - `public Class[] getClasses()`
    - returns an array of objects of type `Class`;
    - all classes and interfaces, public members of the class represented by this `Class` object

The class `Class`

- Methods (cont'd)
  - `Field[] getFields`
    - returns an array containing `Field` objects reflecting all the accessible public fields of the class or interface represented by this `Class` object
  - `Method[]getMethods()`
    - returns an array containing `Method` objects reflecting all the public *member* methods of the class or interface represented by this `Class` object, including those declared by the class or interface and those inherited from superclasses and superinterfaces.
  - `Constructor[]getConstructors()`
    - returns an array containing `Constructor` objects reflecting all the public constructors of the class represented by this `Class` object.

The `instanceof` Operator

- The `instanceof` operator checks if an object is of the type given as its second argument

  `Object instanceof ClassName`

  - This will return `true` if `Object` is of type `ClassName`, and otherwise return `false`
  - Note that this means it will return `true` if `Object` is the type of *any descendent class* of `ClassName`

The `getClass()` Method

- Every object inherits the same `getClass()` method from the `Object` class
  - This method is marked `final`, so it cannot be overridden
- An invocation of `getClass()` on an object returns a representation *only* of the class that was used with `new` to create the object
  - The results of any two such invocations can be compared with `==` or `!=` to determine whether or not they represent the exact same class

```
(object1.getClass() == object2.getClass())
```

`instanceof` and `getClass()`

- Both the `instanceof` operator and the `getClass()` method can be used to check the class of an object
- However, the `getClass()` method is more exact
  - The `instanceof` operator simply tests the class of an object
  - The `getClass()` method used in a test with `==` or `!=` tests if two objects *were created with* the same class

Examples

- Print a class name using a `Class` object

```
void printClassName(Object obj) {
   System.out.println(obj + " is of class " +
   obj.getClass().getName());
}
```

- More examples

```
Circle c = new Circle(5);
printClassName(c);
Class c1 = c.getClass();
System.out.println(c1.getName()); // prints "Circle"
Triangle t = new Triangle(7);
printClassName(t);
try {
   Class c2 = Class.forName("Triangle");
   System.out.println(c2.getName()); // prints "Triangle"
}
catch (ClassNotFoundException e) {
    System.err.println("No class for \"Triangle\"" +
  e.getMessage());
}
```

Summary

- Inheritance
  - base class, superclass
  - derived class, subclass
  - hierarchy
  - **super** and **this** constructors
  - access to instance variables
  - method overriding vs overloading
- Interfaces, abstract classes, concrete classes
- Polymorphism
- The class `Object`
  - everything inherits from it
  - methods to override
- The class `Class`
  - useful methods
- The `instanceof` operator vs `getClass()` method