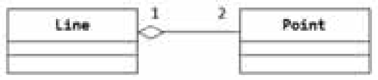
CompSci 230 S2
Object Oriented Software Development

Static & Dynamic Binding

Review Quizzes

```
public class Point {
  private int x;    // x co-ordinate
   private int y;    // y co-ordinate
   public int getX() {......}
   public int getY() {......}
   …
```

- composition



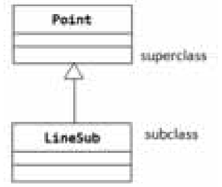
```
public class Line {
   // A line composes of two points (as instance variables)
   private Point begin;    // beginning point
   private Point end;      // ending point

  public MyLine(int x1, int y1, int x2, int y2) {
      begin = new MyPoint(x1, y1);  // Construct the instances declared
      end   = new MyPoint(x2, y2);
   }
  public double getLength() {
      return begin.distance(end);  // Point's distance()
   }
...
```

Review Quizzes

```
public class Point {
   private int x;    // x co-ordinate
   private int y;    // y co-ordinate
   public int getX() {......}
   public int getY() {......}
  …
```

- Inheritance



```
public class LineSub extends Point {
   // A line needs two points: begin and end.
   // The begin point is inherited from its superclass Point.
  Point end;               // Ending point

  public LineSub (int beginX, int beginY, int endX, int endY) {
      super(beginX, beginY);             // construct the begin Point
      this.end = new Point(endX, endY);  // construct the end Point
   }
   public double getLength() {
         return super.distance(end);  // Point's distance()
   }
```

Review Quizzes

- Look at the following code. The method in line _______ will override the method in line _______.
  - Line 10, line 4
  - Line 11, line 5
  - None of the above

```
Line  1   public class ClassA
Line  2   {
Line  3     public ClassA() {}
Line  4     public int method1(int a){}
Line  5     public double method2(int b){}
Line  6   }
Line  7   public ClassB extends ClassA
Line  8   {
Line  9     public ClassB(){}
Line 10     public int method1(int b, int c){}
Line 11     public double method2(double c){}
Line 12   }
```

Review Quizzes

- Look at the following code.
- Which method1 will be executed as a result of the following statements?
  - ClassA item1 = new ClassC();
  - item1.method1();

```
Line  1   public class ClassA
Line  2   {
Line  3     public ClassA() {}
Line  4     public void method1(){}
Line  5   }
Line  6   public class ClassB extends ClassA
Line  7   {
Line  8     public ClassB(){}
Line  9     public void method1(){}
Line 10   }
Line 11   public class ClassC extends ClassB
Line 12   {
Line 13     public ClassC(){}
Line 14     public void method1(){}
Line 15   }
```

  - Line 4
  - Line 9
  - Line 14
  - This is an error and will cause the program to crash

Agenda & Reading

- Topics:
  - Static & Dynamic Binding
  - Casting
- Reading
  - Java how to program Late objects version (D & D)
    - Chapter 10
  - The Java Tutorial
    - Creating and Using Packages
      - http://docs.oracle.com/javase/tutorial/java/package/packages.html
    - Controlling Access to Members of a Class
      - http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html
    - Overriding and Hiding Methods
      - http://docs.oracle.com/javase/tutorial/java/IandI/override.html
    - Hiding Fields
      - http://docs.oracle.com/javase/tutorial/java/IandI/hidevariables.html

1.Introduction

- If you have more than one method of same name (method overriding) or two variable of same name in same class hierarchy it gets tricky to find out which one is used during runtime as a result of there reference in code.
- This problem is resolved using **static and dynamic binding** in Java.
- Binding is the process used to link which **method** or **variable** to be **called** as result of the reference in code.
  - Most of the references is resolved during **compile time** but some references which depends upon **Object** and **polymorphism** in Java is resolved during **runtime** when actual object is **available**.
  - When a method call is **resolved** at **compile** time, it is known as **static binding**, while if method invocation is **resolved** at **runtime**, it is known as **Dynamic binding** or **Late binding**.

1.Introduction

- **private**, **final** and **static methods** and variables uses static binding and resolved by compiler because compiler knows that they can't be overridden and only possible methods are those, which are defined inside a class, whose reference variable is used to call this method.
- Static binding uses **Type information** for binding while Dynamic binding uses **Object** to resolve binding.
- Static Binding
  - Variables – static binding
    - Variables are resolved using static binding which makes there execution fast because no time is wasted to find correct method during runtime.
  - Private, final, static methods – static binding
  - Overloaded methods are resolved using static binding
- Dynamic Binding
  - Overridden methods are resolved using **dynamic binding** at runtime.

2.Static Typing Restrictions

```
Python
x = 10;
x = "Hello"
```
✓ ✓

```
Java
int x = 10;
x = "Hello";
```
✓ ✗

- Java is a "type-safe" language.
  - compile-time checks are carried out to determine whether variable usage is **valid**
    - Every variable name is bounded both to a **static** type (at compile time, by means of a data **declaration**) and to an object/null. The binding to an object is optional.

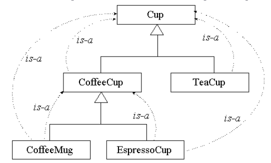
```
Cup c1 = new Cup();
Cup c2 = null;
```

```
Cup c3;
```

- The type imposes restrictions on how a variable can be used.
  - The type restricts the classes of object to **which** the variable can refer and the **message that can be sent** using the variable

2.Static Typing Restrictions Variables

- A reference variable of static type T can refer to an instance of class T or to an instance of any of T's subclasses
  - The is-a relationship applies only up the hierarchy from a subclass to its direct (and indirect) superclasses, and not down the hierarchy.



```
Cup c = new EspressoCup();
CoffeeCup c1 = new EspressoCup();
EspressoCup c2 = new EspressoCup();
Cup c3 = new CoffeeMug();
CoffeeCup c4 = new CoffeeMug();
CoffeeMug c5 = new CoffeeMug();
Cup c6 = new TeaCup();
```
✓

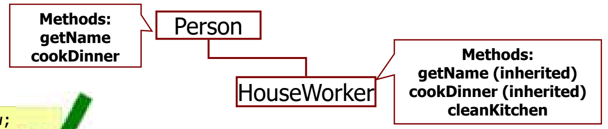
- A superclass object cannot be treated as a subclass object, because a superclass object is **NOT** an object of any of its subclasses.
  - Subclass can have additional subclass only **members**. Therefore, assigning a superclass reference to a subclass variable is not allowed

```
Person p = new Person("Dick", "Smith");
HouseWorker w = p;
```
✗

```
error: incompatible types: Person cannot be converted to HouseWorker
```

2.Static Typing Restrictions Messages

- Through a reference variable of static type T, the set of messages that can be sent using that variable are the methods defined by class T and its superclasses.
  - For example:



```
Person p = w;
p.getName()
p.cookDinner()
```
✓

```
p.cleanKitchen();
```
✗ Compile-time error

```
HouseWorker w = new HouseWorker("Theresa", "Thompson");
w.cleanKitchen();
w.getName();
w.cookDinner();
```
✓

3.Static Binding – Variables Instance variables

- Static binding only uses Type information
  - Access to fields is governed by the **type** of the reference
- For example:

```
class Base {
  public int x = 10;
}
```

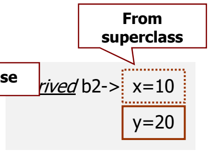
```
public class Derived extends Base {
  public int y = 20;
}
```

- Case 1: a Derived object with a **Derived** reference

```
//Case 1:
Derived b1 = new Derived();
System.out.println("b1.x=" + b1.x);
System.out.println("b1.y=" + b1.y);
```

instance variable x : inherited from Base

instance variable y in Derived

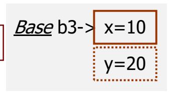


- Case 2: a Derived object with a **Base(superclass)** reference

```
//Case 2:
Base b2 = new Derived();
System.out.println("b2.x=" + b2.x);
//System.out.println("b2.y=" + b2.y);
```
✗

instance variable x : inherited from ***Base***,

There is no y declared in ***Base*** class -> compile error!



3.Static Binding – Variables
Two variable of same name

- Static binding only uses Type information
  - Access to fields is governed by the **type** of the reference
- For example:

```
class Base2 {
  public int x = 10;
}
```

```
public class Derived2 extends Base2 {
  public int x = 20;
}
```

  - a field that has the same name as a field in the superclass hides the superclass's field
- Case 1: a Derived object with a Derived reference

Derived b2-> x=10
x=20

```
//Case 1:
Derived2 b1 = new Derived2();
System.out.println("b1.x=" + b1.x);
```

instance variable x from *Derived (20)*

- Case 2: a Derived object with a Base(superclass) reference

b3 is declared as type Base

instance variable x from *Base*, (10)

```
//Case 3:
Base2 b2 = new Derived2();
System.out.println("b2.x=" + b2.x);
```

Base b3-> x=10
x=20

3.Static Binding – Variables
Get the hiding values

- Within the subclass, the field in the superclass cannot be referenced by its simple name
- To "unshadow" it by referring to super.x

```
public class Derived extends Base {
  ...
  public void method1() {
    System.out.println("x from Derived:" + x);
    System.out.println("x from superclass: " + super.x);
    ...
```

method In the subclass

  - Note:
    - this.x access the field name x, defined in the child class
    - super.x access the field name x, defined in a parent class
    - super.super.x is not valid
    - x was not specified as private.

3.Static Binding – Variables
Static Variables

- Static binding only uses Type information
  - Access to fields is governed by the type of the reference
  - Class variable: one copy only, all instances of the class share the same static variable
  - Example:

```
class Base3 {
  public static int x=10;
}
```

```
public class Derived3 extends Base3 {
  public static int y=20;
}
```

    - Invoke the static variable in the Base3 class
    - Invoke the static variables in the Derived3 class

class variables: one copy only

Base: x= 10
Derived: y=20

```
System.out.println("Base3.x=" + Base3.x);

System.out.println("Derived3.y=" + Derived3.y);
System.out.println("Derived3.y=" + Derived3.x);
```

```
Base3.x=10
Derived3.y=20
Derived3.y=10
```

3.Static Binding
private, final and static methods

- A final method in a superclass cannot be overridden in a subclass.
  - Methods that are declared **private** are implicitly **final**, because it's not possible to override them in a subclass.
  - Methods that are declared **static** are implicitly **final**.
- A final method's declaration can never change, so all subclasses use the **same method implementation**, and calls to final methods are resolved at compile time—this is known as static binding.

3.Static Binding
static Methods

- Only the non-static methods of a class can be overridden
  - Example:

```
class Base4 {
  public static int x = 10;
  public static void get() {
    System.out.println("static:Base:get");
  }
}
```

```
public class Derived4 extends Base4 {
  public static int x = 20;
  public static void get() {
    System.out.println("static:Derived:get");
  }
}
```

    - Invoke the static method in the Base4 class
    - Invoke the static method in the Derived4 class

```
Base4.get();

Derived4.get();
```

```
static:Base:get
static:Derived:get
```

3.Static Binding
private Methods

- When you declare a method in a Java class, you can allow or disallow other classes and object to call that method.

```
class Base5 {
  private void get() {
    System.out.println("private:Base:get");
  }
}
```

```
public class Derived5 extends Base5 {
  private void get() {
    System.out.println("private:Derived:get");
  }
 public static void main(String[] args) {
    Derived5 b1 = new Derived5();
    b1.get();
  }
}
```

```
private:Derived:get
```

3.Static Binding
final Methods

- You use the final keyword in a method declaration to indicate that the method cannot be overridden by subclasses.

```
class Base6 {
  public final void get() {
    System.out.println("final:Base:get");
  }
}

public class Derived6 extends Base6 {
  /*
   public void get() {
     System.out.println("final:Derived:get");
  }
  */
}
```

```
error: get() in Derived6 cannot override get() in Base6
```

3.Static Binding
Overloading Methods

- The practice of defining two or more methods **within the same class** that share the **same** name but have different signatures is called overloading methods.
  - The signature of a method is comprised of its name, its parameter types and the order of its parameters.
  - The signature of a method is not comprised of its return type nor its visibility.

```
class Base7 {
  public void aMethod() {
    System.out.println("called aMethod()");
  }
  public void aMethod(int x) {
    System.out.println("called aMethod(x)");
  }
  public static void main(String[] args) {
    Base7 b1 = new Base7();
    b1.aMethod(8);
  }
}
```

```
called aMethod(x)
```

4.Dynamic Binding

- Dynamic Binding refers to the case where compiler is not able to resolve the call and the binding is done at runtime only.
  - Java compiler can't be sure of what type of object this reference would be pointing to at runtime.
  - Polymorphism in Java is resolved during runtime when actual object is available.
- Example:

```
class Base8 {
  public void g() {
    System.out.println("Base:g");
  }
}
```

```
public class Derived8 extends Base8 {
  public void g() {
    System.out.println("Derived:g");
  }
}
```

- Case 1: a Derived object with a Derived reference

```
//Case 1:
Derived8 b1 = new Derived8();
b1.g();
```

```
Derived:g
```

g() in Derived?
Invoke g() in Derived

- Case 2: a Derived object with a Base(superclass) reference

```
//Case 3:
Base8 b2 = new Derived8();
b2.g();
```

```
Derived:g
```

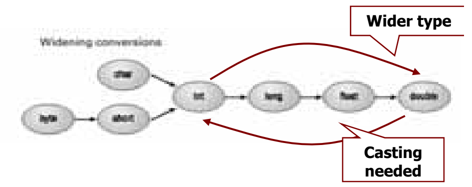
g() in Base?
Invoke g() in Derived

5.Casting

- **Casting** an object to a different type will **permit** access to fields shadowed by the type of the original object
- Casting an object to a different type will have **NO** effect on which method is invoked in response to a given message. Once overridden, methods remain overridden
- To cast, just precede a value with the parenthesised name of the desired type
  - A **ClassCastException** is thrown if the type cast is incompatible with the type of object to be pointed to.
  - To check that a potential cast is safe, the instanceof operator should be used
    - The **instanceof** operator Can be used to determine the type of an object
    - Returns **true** if <obj> can be assigned into a reference of type <Class name>. That is, instanceof checks if the <obj> is-an **instance** of the specified class, provided obj is not null.

```
<obj> instanceof <Class name>
```

5.Casting Primitive Types



- Widening conversions
  - Wider assignment
  - Wider Casting
    - Casting a value to a wider value is always permitted but never required. However, explicitly casting can make your code more readable

```
double d = 4.9;
int i = 10;
double d1, d2;
```

```
d1 = i;
```

Assignment conversion
d1 = 10.0

```
d2 = (double) i;
```

Casting conversion
(optional)
d2 = 10.0

- Narrowing conversion
  - Narrow assignment – Compile-time error
  - Narrow Casting – Loss of information
    - You must use an explicit cast to convert a value in a large type to a smaller type, or else converting that value might result in a loss of precision.

```
double d = 4.9;
int i = 10;
int i1, i2;
```

```
//i1 = d;
```

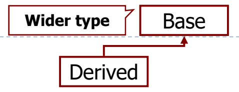
Assignment conversion
Compile-time error

```
i2 = (int) d;
```

Narrow Casting: i2=4
NOTE: Everything beyond the decimal point will be truncated

5.Casting Object Types

Example: casting/CastObject.java



- Widening conversions
  - Wider object reference assignment conversion
  - Wider object reference casting (optional)

```
Base b = new Base();
Derived d = new Derived();
Base b1, b2;
System.out.println(d.y);
```

```
b1 = d;
//System.out.println(b1.y);
```

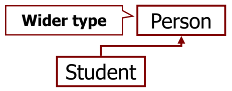
Assignment conversion - OK
However, no access to fields in Derived

```
b2 = (Base) d;
//System.out.println(b2.y);
```

Widening Casting conversion - OK
However, no access to fields in Derived

- Note:
  - Reference conversion, like primitive conversion, takes place at compile time, because the compiler has all the information it needs to determine whether the conversion is legal
  - Inheritance relation is an “is a” relation; the parent object is more general than child object
  - The general rule of thumb is that converting to a superclass is permitted (because super class is wider than sub class)

5.Casting Object Types



Example: casting/CastObject.java

- Narrowing conversion
  - Narrow object reference assignment – Compile-time error
  - Narrow object reference casting –
    - Compile-time OK,
    - Run-time?
      - Like primitive casting: By using a cast, you convince the compiler to let you do a conversion that otherwise might not be allowed
      - The run-time check determines whether the class of the object being cast is compatible with the new type

```
Base b = new Base();
Derived d = new Derived();

Derived d1, d2, d3;
```

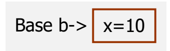
```
//d1 = b;
```

Compile-time error

```
d2 = (Derived) b;
```

Compile-time OK, Run-time ERROR
b is an instance of class Base, not Derived

```
java.lang.ClassCastException: Base
```

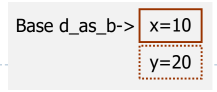


```
Base d_as_b = new Derived();
```

```
d3 = (Derived) d_as_b;
```

Compile-time OK, narrow casting, casting from superclass to subclass

Run-time OK
d_as_b is an instance of Derived



Using Casting

Example: Casting/Recover.java

- Shadowing
  - Using type casting, it is possible to recover a hidden member (class variable, class method, or an instance variable), but not an overridden method (i.e. an instance method). An overridden method can be recovered only by means of a super call in derived class

```
class Base2 {
  public int x = 10;
}
```

```
public class Derived2 extends Base2 {
  public int x = 20;
}
```

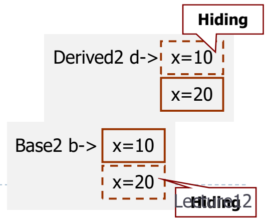
```
Derived2 d = new Derived2();
Base2 b = new Derived2();

System.out.println( "d.x=" + d.x);
System.out.println( "((Base2)d).x=" + ((Base2)d).x);
```

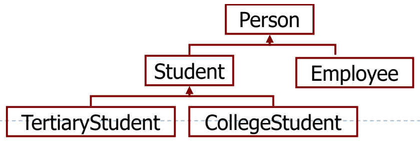
```
20
10
```

```
System.out.println( "b.x=" + b.x);
System.out.println( "((Derived2)b).x=" + ((Derived2)b).x);
```

```
10
20
```



5.Casting Rules



- At compile-time, checking is carried out to determine whether the cast-type and the type of the existing reference variable are related through inheritance.
- At run time, a check is made to determine whether the object actually pointed to is compatible with the cast-type.

```
Student s1 = new TertiaryStudent();
TertiaryStudent s2 = new TertiaryStudent();

Employee e1 = new Employee();

Student s3 = new CollegeStudent();
Student s4 = new Student();
```

	Compile-time	Run-time
(TertiaryStudent) s1	OK	OK
(TertiaryStudent) s2	OK	OK
(TertiaryStudent) e1	ERROR	
(TertiaryStudent) s3	OK	ERROR
(TertiaryStudent) s4	OK	ERROR
(Students) S2	OK	OK

Summary

```
Student s = new Student();
Person p = s;
```

- Object Conversion
  - Make it possible to reference an object in a different way by assigning it to an object reference of a different type.
  - It is handled automatically by the compiler.
  - The object is NOT converted or changed in any way. Once instantiated, an object NEVER changes its type (a Cat will always be a Cat). The concept is similar to the "widening conversions" performed on primitive data types.
- Casting
  - It is needed when the compiler doesn't recognize an automatic conversion. The concept is similar to the casting of primitive data types to perform a "narrowing conversion".
  - Casting does not change the reference or the object being pointed to. It only changes the compiler's treatment of the reference
  - Casting is only legal between objects in the same inheritance hierarchy
  - You can safely cast from an object to its superclass

Exercise 1

- What is the output?

```
Base01 b = new Base01();
Derived01 d = new Derived01();

System.out.println("b.x = " + b.x);
System.out.println("b.y = " + b.y);
System.out.println("b.foo() = " + b.foo());
System.out.println("b.goo() = " + b.goo());
```

```
System.out.println("d.x = " + d.x);
System.out.println("d.y = " + d.y);
System.out.println("d.foo() = " + d.foo());
System.out.println("d.goo() = " + d.goo());
```

```
System.out.println("d.x2 = " + d.x2);
System.out.println("d.y2 = " + d.y2);
System.out.println("d.foo2() = " + d.foo2());
System.out.println("d.goo2() = " + d.goo2());
```

Structure

- Base01 & Derived01

```
class Base01 {
  public int x = 10;
  static int y = 10;
  Base01() {
    x = y++;
  }
  public int foo() {
    return x;
  }
  public static int goo() {
    return y;
  }
}
```

```
class Derived01 extends Base01 {
  int x2= 20;
  static int y2 = 20;
  Derived01() {
    x2 = y2++;
  }

  public int foo2() {
    return x2;
  }
  public static int goo2() {
    return y2;
  }
}
```

Overriding, hiding, and overloading methods

- An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass *overrides* the superclass's method.
- If a subclass defines a class method with the same signature as a class method in the superclass, the method in the subclass *hides* the one in the superclass.
  - The distinction between hiding and overriding has important implications.
    - The version of the overridden method that gets invoked is the one in the subclass.
    - The version of the hidden method that gets invoked depends on whether it is invoked from the superclass or the subclass.
- Overloaded methods are differentiated by the number and the type of the arguments passed into the method.
  - The compiler does not consider return type when differentiating methods, so you cannot declare two methods [in the same class] with the same signature even if they have a different return type.

Exercise 2

- Base2, Derived2

```
class Base02 {
  public int x = 10;
  static int y = 10;
  Base02() {
    x = y++;
  }
  public int foo() {
    return x;
  }
  public static int goo() {
    return y;
  }
}
```

```
class Derived02 extends Base02 {
  int x= 20;   // shadow
  static int y = 20;
  Derived02() {
    x = y++;
  }

  public int foo() { // overrding
    return x;
  }
  public static int goo() {
    return y;
  }
}
```

Exercise 2

- What is the output?

```
Derived02 d = new Derived02();
System.out.println("The Derived object");
System.out.println("d.x = " + d.x);  // hide
System.out.println("d.y = " + d.y);
System.out.println("d.foo() = " + d.foo()); // override
System.out.println("d.goo() = " + d.goo());
Base02 b = new Derived02();
System.out.println("\nThe Derived object");
System.out.println("b.x = " + b.x);
System.out.println("b.y = " + b.y);
System.out.println("b.foo() = " + b.foo()); // override
System.out.println("b.goo() = " + b.goo());
```

```
System.out.println("((Base02)b).x = " + ((Base02)b).x);
System.out.println("((Derived02)b).x = " + ((Derived02)b).x);
System.out.println("\n((Base02)b).y = " + ((Base02)b).y);
System.out.println("((Derived02)b).y = " + ((Derived02)b).y);
System.out.println("\n((Base02)b).foo()=" + ((Base02)b).foo());
System.out.println("((Derived02)b).foo()="+ ((Derived02)b).foo());
System.out.println("\n((Base02)b).goo()=" + ((Base02)b).goo());
System.out.println("((Derived02)b).goo()="+ ((Derived02)b).goo());
```

Exercise 3

- What is the output?

```
Derived04 b = new Derived04();
Base04 a = (Base04) b; // casting b to Base04's instance
System.out.println( a.i ); // refer to Base04.i
System.out.println( a.f() );
System.out.println( a.i );
System.out.println( a.f() );
```

```
class Base04 {
  int i = 1;
  public int f() {
    return i;
  }
}
```

```
class Derived04 extends Base04 {
  int i = 2; // shadows i in Base04
  public int f() { // override f() in Base04
    return -i;
  }
}
```

Exercise 4

- A call of an overridden method can only be resolved at runtime as the compiler can't be sure of what type of object this reference would be pointing to at runtime.
  - The search for a method begins with the dynamic class.
    - If this class doesn't implement the method (i.e. it doesn't introduce or override the method), the search progresses to it's superclass. This process is repeated up the hierarchy until the method is found.

```
class Base {
  public void f() {}
  public void g() {}
}
```

```
public class Derived extends Base {
  public void g{} { ... }
  public void h() {…}
}
```

```
Derived b2 = new Derived();
Base b3 = new Derived();
```

Dynamic class

`b2.f();` — f() in Derived, f() in Base? Invoke f() in ____

`b2.g();` — g() in Derived? Invoke g() in ___

`b2.h();` — h() in Derived? Invoke h() in ___

`b3.f();` — f() in Base? Invoke f() in ___

`b3.g();` — g() in Base? Invoke g() in ____

`b3.h();` — h() in Base? Compile time error