Inheritance 

1. Overview

The learning objectives of this laboratory session are:

- Understand and properly use the superclass constructors
- Understand overloading and overriding of methods
- Understand and learn how to avoid common problems with inheritance

2. super(...) - The superclass (parent) constructor

An object has the fields of its own class plus all fields of its parent class, grandparent class, all the way up to the root class Object.
It is necessary *to initialize all fields*, therefore *all constructors must be called*!
The Java compiler automatically inserts the necessary constructor calls in the process of constructor chaining, or you can do it explicitly.
The Java compiler inserts a call to the parent constructor (**`super`**) if you don't have a constructor call as the first statement of you constructor. Given the following code

```
public class Point {
    int m_x;
    int m_y;

    //============ Constructor
    public Point(int x, int y) {
        m_x = x;
        m_y = y;
    }

    //============ Parameterless default constructor
    public Point() {
        this(0, 0);  // Calls other constructor.
    }
    . . .
}
```

The following is the equivalent of the constructor above.

```
//============ Constructor (same as in above example)
public Point(int x, int y) {
    super();  // Automatically done if you don't call constructor here.
    m_x = x;
    m_y = y;
}
```

2.1 Why you might want to call super explicitly

Normally, you won't need to call the constructor for your parent class because it's automatically generated, but there are two cases where this is necessary.

1. You want to call a parent constructor which has parameters (the automatically generated super constructor call has no parameters).
2. There is no parameterless parent constructor because only constructors with parameters are defined in the parent class.

Every object contains the instance variables of its class. What isn't so obvious is that every object also has all the instance variables of all super classes (parent class, grandparent class, etc). These super class variables must be initialized before the class's instances variables.

2.1.1 Automatic insertion of super class constructor call

When an object is created, it's necessary to call the constructors of all super classes to initialize their fields. Java does this automatically at the beginning if you don't.
For example, the first Point constructor could be written

```
public Point(int xx, int yy) {
    super();  // Automatically inserted
    x = xx;
    y = yy;
}
```

2.1.2 Explicit call to superclass constructor

Normally, you don't explicitly write the call of the constructor for your parent class, but there are two cases where this is necessary:
Passing parameters. You want to call a parent constructor which has parameters (the default construct has no parameters). For example, if you are defining a subclass of **`JFrame`** you might do the following.

```
class MyWindow extends JFrame {
    . . .
    //======== constructor
    public MyWindow(String title) {
        super(title);
        . . .
```

In the above example you wanted to make use of the **`JFrame`** constructor that takes a title as a parameter. It would have been simple to let the default constructor be called and use a setter method as an alternative.

```
class MyWindow extends JFrame {
    . . .
    //======== constructor
    public MyWindow(String title) {
        // Default superclass constructor call automatically inserted.
        setTitle(title);  // Calls method in superclass.
        . . .
```

No parameterless constructor. There is no parent constructor with no parameters. Sometimes is doesn't make sense to create an object without supplying parameters. For example, should there really be a **`Point`** constructor with no parameters? Although the previous example did define a parameterless constructor to illustrate use of **`this`**, it probably isn't a good idea for points.

2.1.3 Example of class without parameterless constructor

```
////////////////// class without a parameterless constructor.
//  If any constructor is defined, the compiler doesn't
//  automatically create a default parameterless constructor.
class Parent
{
    int _x;
    Parent(int x) {   // constructor
        _x = x;
    }
```

```
}

////////////////// class that must call super in constructor
class Child extends Parent
{
    int _y;
    Child(int y) {  // WRONG, needs explicit call to super.
        _y = y;
    }
}
```

In the example above, there is no explicit call to a constructor in the first line of constructor, so the compiler will insert a call to the parameterless constructor of the parent, but there is no parameterless parent constructor! Therefore this produces a compilation error. The problem can be fixed by changing the **Child** class.

```
////////////////// class that must call super in constructor
class Child extends Parent
{
    int _y;
    Child(int y) {
        super(0);
        _y = y;
    }
}
```

Or the **Parent** class can define a parameterless constructor.

```
////////////////// class without a parameterless constructor.
//  If any constructor is defined, the compiler doesn't
//  automatically create a default parameterless constructor.
class Parent
{
    int _x;
    Parent(int x)
    {   // constructor with parameter
        _x = x;
    }

    Parent()
   {        // constructor without parameters
        _x = 0;
    }
}
```

A better way to define the parameterless constructor is to call the parameterized constructor so that any changes that are made only have to be made in one constructor.

```
    Parent()
    {        // constructor without parameters
        this(0);
    }
}
```

Note that each of these constructors implicitly calls the parameterless constructor for its parent class, etc, until the **`Object`** class is finally reached.

2.1.4 How do you override a method?

To override a method in your new class, simply reproduce the name, argument list, and return type of the original method in a new method definition in your new class. Then provide a body for the new method. Write code in that body to cause the behavior of the overridden method to be appropriate for an object of your new class.
Here is a more precise description of method overriding taken from the excellent book entitled The Complete Java 2 Certification Study Guide, by Roberts, Heller, and Ernest:

> *"A valid override has identical argument types and order, identical return type, and is not less accessible than the original method. The overriding method must not throw any checked exceptions that were not declared for the original method."*

Any method that is not declared **`final`** can be overridden in a subclass.

2.1.5 Overriding versus overloading

Don't confuse method overriding with method overloading. Here is what Roberts, Heller, and Ernest have to say about overloading methods:

> *"A valid overload differs in the number or type of its arguments. Differences in argument names are not significant. A different return type is permitted, but is not sufficient by itself to distinguish an overloading method."*

3 Common problems with inheritance

3.1 Variable shadowing

When both a parent class and its subclass have a field with the same name, this technique is called variable shadowing. If the field in the parent class has **`private`** access or is in another package and has default access, there is no room for confusion. The child class cannot access the field in question of the parent class. So it's clear with which variable any reads and writes in the child class will take place. However, if the like-named variable in the parent class is accessible to instances of the child class, some rather nonintuitive rules determine which field is accessed by different code invocations. The general rule is that the variable accessed depends on the class to which the variable has been cast.
For example, Listing A includes two classes, **`Base`** and **`Sub`**, which represent a parent class/subclass relationship. Both of these classes have an integer field named **`field`**. Listing B shows the instantiation of class **`Sub`** followed by many legal ways to refer to one or another of the **`field`** variables.

Listing A

```
public class Base
{
    public  int field = 0;
    public int getField() { return field; }
}
public class Sub extends Base
{
    public  int field = 1;
    public int getField() { return field; }
}
```

Listing B

```
Sub s = new Sub();
Base b = s;
System.out.println(s.field); // access one
System.out.println(b.field); // access two
System.out.println(((Sub)b).field); // access three
```

```
System.out.println(((Base)s).field); // access four
```

The first of these techniques, marked `access one`, directly accesses the **`field`** variable in the instance of **`Sub`** named **`s`**. This access, as one would expect, yields a value of 1. `Access two`, however, shows the logical disconnect involved in variable shadowing. Even though the expression **`b==s`** is true, the second access, **`b.field`**, evaluates to 0.
This difference between **`b.field`** and **`s.field`**, despite the referential equivalency of **`b`** and **`s`**, clearly displays the minefield inherent in variable shadowing. The only difference between **`b`** and **`s`** in this example is the type of the variable in which they're stored. Variable shadowing isn't the only case wherein the value of something is determined by the type you call it, but developers nonetheless trip over these issues with surprising regularity.
The expressions marked `access three` and `access four` follow the rule of type dictating value. Specifically, `access three` resolves to a value of 1 because object **`b`** is cast to type **`Sub`** before its **`field`** variable is checked. Similarly, `access four` yields a value of 0 due to object **`s`** being cast to type **`Base`** before the **`field`** is accessed.

3.2 Method overriding

While like-named fields in subclasses overshadow their namesakes in parent classes, different terminology is used to discuss similar problems with methods. When a parent class and a child class each have a method with the same signature, the method of the child class overrides the method of the parent class. In Listing A, one can see that the **`getField`** method in class **`Sub`** has the same name and parameters, which is to say none, as the **`getField`** method in class **`Base`**.

Listing C

```
Sub s = new Sub();
Base b = s;
System.out.println(s.getField); // access one
System.out.println(b.getField); // access two
System.out.println(((Sub)b).getField); // access three
System.out.println(((Base)s).getField); // access four
```

Listing C shows the same object instantiation as found in Listing B and many possible invocations of the two **`getField`** methods. However, unlike Listing B, the results in Listing C are more to the liking of an object-oriented programmer. The line marked `access one` in Listing C invokes the **`getField`** method on a reference to the single object in the example while it is stored in a variable of type **`Sub`**. As was the case in the similar looking access one in Listing B, this invocation yields a value of 1.
The difference between method overriding and variable overshadowing is visible when comparing `access two` in Listing C to its like-named mate in Listing B. Whereas in Listing B the parent class's **`field`** was accessed, in Listing C the **`field`** variable in the subclass is invoked, despite the type of the variable in which the reference is stored, for a resulting value of 1.
This strict adherence to reference/instance identity vs. variable type is the heart of *polymorphism*. Polymorphism is the feature that delegates behavior to the actual class of the referenced instance — not to the type in which the reference is stored. Polymorphism is a method-only affair.
Continuing with the examples, one comes upon access three and access four in Listing C. Both of these produce a value of 1. Again this is due to the polymorphic nature of method access. To access the shadowed variable in the **`Base`** class via method invocation, one must use the syntax **`super.field`** within the **`Sub`** class.

4 Lab Tasks

2.1. Study the text and source code provided in this paper.

2.2. Study and execute the following code:

```
class Insect
{
  private int i = 9;
  protected int j;
  Insect()
  {
    System.out.println("i = " + i + ", j = " + j);
    j = 39;
  }
  private static int x1 =
    printInit("static Insect.x1 initialized");
  static int printInit(String s)
  {
    System.out.println(s);
    return 47;
  }
}
public class Beetle extends Insect
{
  private int k = printInit("Beetle.k initialized");
  public Beetle()
  {
    System.out.println("k = " + k);
    System.out.println("j = " + j);
  }
  private static int x2 =
    printInit("static Beetle.x2 initialized");
  public static void main(String[] args)
  {
    System.out.println("Beetle constructor");
    Beetle b = new Beetle();
  }
} /* Output:
static Insect.x1 initialized
static Beetle.x2 initialized
Beetle constructor
i = 9, j = 0
Beetle.k initialized
k = 47
j = 39
*/
```

Note the order of initialization.

2.3. Add another insect type and then subclasses of Beetle and your type. Add code to show the order of initialization operations for your types.

2.4. Create an inheritance hierarchy of **Rodent**: **Mouse**, **Squirrel**, **Hamster**, etc. In the base class, provide methods that are common to all **Rodent**s, and override these in the derived classes to perform different behaviors depending on the specific type of **Rodent**. Create an array of **Rodent**, fill it with different specific types of **Rodent**s, and call your base-class methods to see what happens.

2.5. Create an **abstract** class with no methods. Derive a class and add a method. Create a **static** method that takes a reference to the base class, downcasts it to the derived class, and calls the method. In **main( )**, demonstrate that it works. Now put the **abstract** declaration for the method in the base class, thus eliminating the need for the downcast.