Binding

COMP360

“No obligation to do the impossible is binding.”

Marcus Tullius Cicero

Reading

Read section 2.2.3 of the textbook

Run Time Binding of Method

- In the following example, a parent class and two child classes all implement the same method
- If an object of the parent class is passed as a parameter, Java cannot tell until execution time which method to execute

Parent Class

```
public class Mammal {
    public int id = 1;

    public void displayID() {
        System.out.println("Mammal "+ id);
    }
}
```

Children Classes

```
public class Dog extends Mammal {
   public void displayID() {
      System.out.println("Dog "+id);
    }
}
public class Cat extends Mammal {
   public void displayID() {
      System.out.println("Cat "+id);
    }
}
```

Passing the Parent as a Parameter

- Another part of the program has a method that takes an object of the Mammal class as a parameter
- This method calls the displayID method on the Mammal object

```
static void showIt( Mammal animal ) {
        animal.displayID();
}
```

Results

```
Mammal  critter  = new Mammal();
Dog     canine = new Dog();
Cat     feline   = new Cat();
critter.displayID();          // displays Mammal 1
canine.displayID();           // displays Dog 1
feline.displayID();           // displays Cat 1
showIt( critter );            // displays Mammal 1
showIt( canine );             // displays Dog 1
showIt( feline );             // displays Cat 1
```

Dynamic Binding from an Interface

- A slightly different version of the program uses a Java interface instead of a parent class
- A parameter can be an interface, although you cannot make objects of an interface

Java Interface

- The interface requires all classes that implement it to have a displayID method

```
public interface RequireDisplay {
    public void displayID();
}
```

Implementing the Interface

```
public class Dog implements RequireDisplay {
   public void displayID() {
      System.out.println("Dog ");
   }
}
public class Cat implements RequireDisplay {
   public void displayID() {
      System.out.println("Cat ");
   }
}
```

Passing the Parent as a Parameter

- Another part of the program has a method that takes an object as a parameter that implements RequireDisplay
- This method calls the displayID method on the object implementing RequireDisplay

```
static void showIt( RequireDisplay animal ) {
        animal.displayID();
}
```

Results

```
Dog  canine = new Dog();
Cat    feline  = new Cat();
canine.displayID();          // displays dog
feline.displayID();          // displays cat
showIt( canine );            // displays dog
showIt( feline );            // displays cat
```

Variable Scope

- The *"Scope"* of a variable refers to where you can use a variable
- Where you declare a variable determines where it can be used
- The variable names used in a method are a completely separate set of names from the names used in any other method, including the main method

Variable Range

- Variables defined in a block can only be used in that block following the variable declaration

```
{
        // you cannot use the variable moth here
        double moth = 72.5;
        // you may use the variable moth here
        // this is the scope of moth
}
// The variable moth is not allowed here
```

Out of Scope

```
int cat;
if (whatever == 0) {
    int dog = 1;
} else {
    int dog = 2;
}
cat = dog * 3;    // Error, dog is out of scope
```

This works

```
int cat, dog;
if (whatever == 0) {
    dog = 1;
} else {
    dog = 2;
}
cat = dog * 3;    // Only one dog variable
```

Method Parameter Scope

- The parameters defined in a method header can be used throughout that method
- Method parameter variables cannot be used in another method

```
void aMethod( int dog, double cat ) {
You can use dog and cat throughout this method
}
```

Declarations First

- Java and C++ allow you to put variable declarations anywhere in a program
- It is usually advantageous to declare a variable at the beginning of a block
- Some older programming languages required you to put all variable declarations at the beginning of a block
  - Fortran – All declarations are before executable statements
  - Cobol – declarations are in the “data division”

for Loop Scope

- A loop counter variable only has scope in the loop

```
for (int cow = 0; cow < 28; cow++) {
        // loop body
        // the loop counter, cow, can be used here
}
// You cannot use cow after the loop
```

Local Variables

```
public static void main( ) {
   double       cat = 5, bird = 47;
   cat = myfunc( bird );
   System.out.print(cat);
}
```

Scope of cat and bird

```
double myfunc(double cow) {
   double bull;
   bull = cow * 2.0;
   return bull;
}
```

Scope of cow and bull

Multiple Variable with the Same Name

- In different parts of a program, you can use the same variable name, but it will mean a different variable
- This can be confusing and should be avoided

Local Variables *(Tricky)*

```
public static void main( ) {
   double       cat = 5, bird = 47;
   cat = myfunc( bird );
   System.out.print(cat);
}
```

Scope of cat and bird

```
int  myfunc(double cow) {
   int    bird;
   bird = (int)cow * 2.0;
   return bird;
}
```

Scope of cow and bird (different bird)

Variables

- A C++ or Java method can use three different types of variables
  - local variables defined in the method
  - parameter variables
  - object instance variables
- Object instance variables can be used in any of the methods of the class

Name Collision

```
public class Collide {
 int rat = 3;
 public int square( int rat ) {
    return rat * rat;
 }
}
```

```
Collide thing = new Collide();
int turtle = thing.square( 2 );
```

- turtle is set to 4

Variable Priority

- A method cannot have a local variable the same name as a parameter variable
- If a class instance variable has the same name as a local variable or parameter, the method will always use the local variable or parameter

More Name Collisions

```
public class Collide {
 int rat = 3;
 public int square( int mouse)
 {
    int rat = mouse * mouse;
    return rat;
 }
}
```

- The class instance variable rat is a different memory location from the local variable rat

Name Collision

```
public class Collide {
 int rat = 3;
 public int square( int rat ) {
    return this.rat * rat;
 }
}
Collide thing = new Collide();
int turtle = thing.square( 2 );
```

- turtle gets the value 6

Separate Compilation

- Java, C++ and many languages allow modules to be compiled separately
- Rules for how variables work with separate compilation are messy
- In C++, **static** on a function or variable outside a function or **private** in Java means it is usable only in the current source file
- This **static** is a different notion from the **static** variables inside a function

Created Elsewhere

- **extern** on a C++ variable or function means that it is declared in another source file
- In Java and C++, method headers without bodies are extern by default