Over-Riding Methods

- If we want to replace a method in a parent class we can create a new one to over-ride it
- Must have **same method signature**: exact same access, name and list of parameters for input or output
- For example if the parent contains a method:
  ```java
  public void doX( int y ) {...}
  ```
  The child can over-ride it by declaring own method with:
  1. Same name: doX()
  2. Same parameter type: int
  3. Same output type: void
  4. Same access: public

An Example of Over-Riding

- Here, the B class over-rides `sum()` method from class A
- What happens when following code is executed?

```java
public class A {
  protected int i, j;
  public A int a, int b ) {
    i = a;
  }
  public int sum(int x) {
    return x + i + j;
  }
  public class B extends A {
    protected int k;
    public B int a, int b, int c) {
      super(a, b);
      k = c;
    }
    public int sum(int x) {
      return x + i + j + k;
    }
  }
}
```

An Example of Overriding, II

- Note that, as usual, when we over-ride a method, the **name** of the input parameter variables we use does not matter (as long as it is **legal** in class we are writing)
- All that really matters is the **type** of those variables, which must occur in **same order** as in the original method

```java
public class A {
  protected int i, j;
  public A int a, int b ) {
    i = a;
  }
  public int sum(int x) {
    return x + i + j;
  }
  public class B extends A {
    protected int k;
    public B int a, int b, int c) {
      super(a, b);
      k = c;
    }
    public int sum(int x) {
      return x + i + j + k;
    }
  }
}
```

This version over-rides too, just like the version on the previous slide, since it uses a single integer parameter (x or y) in both cases
Over-Loading Methods

- Normally, we can not have two members of a Java class using the same name in the same scope.
- For instance, the following two lines of code cannot occur together in the same method or other scope:

```
private int x = 3;
private double x = 2.0;
```

- For method naming, things are a little more flexible.
- The rule is that we cannot have two methods of the same name, if they have the same input parameter types.

An Example of Over-Loading

- Here, the A class has three different versions of plus().
- Each version is allowed, since each can be differentiated based on the actual inputs used when calling the methods at run-time.

```java
public class A {
    public int plus(int x) {
        return x + 1;
    }
    public int plus(int x, int y) {
        return x + y + 1;
    }
    public double plus(int x, double y) {
        return x + y + 1;
    }
}
```

```java
class Example {
    public static void main(String[] args) {
        A a = new A();
        int n1 = a.plus(3);
        int n2 = a.plus(3, 4);
        double n3 = a.plus(3, 8.0);
    }
}
```

First version: returns 4
Second version: returns 12
Third version: returns 12.0

Over-Loading Uses Inputs, Not Output

- When we over-load a method, it is the input types that are used to tell the methods apart, since we will often only have that information at run-time.
- This version of the class A will not compile!

```java
public class A {
    public void printIt(char c) {
        System.out.println("char: " + c);
    }
    public void printIt(int i) {
        System.out.println("int: " + i);
    }
    public void printIt(double n) {
        System.out.println("double: " + n);
    }
}
```

```java
class Example {
    public static void main(String[] args) {
        A a = new A();
        a.printIt('c');
        a.printIt(3);
        a.printIt(3.0);
    }
}
```

First version: uses char
Second version: uses int
Third version: uses double

Over-Loading Uses Smallest (Most Specific) Types Possible

- If we over-load a method, using types that are sometimes convertible (like int → double), the compiler/JVM will always use the smallest type it can.
- It will never automatically convert to a higher type, if it doesn’t have to.
- If we want to use converted types, we need to do the casting ourselves.

```java
public class A {
    public void printIt(int i) {
        System.out.println("int: " + i);
    }
    public void printIt(int i) {
        System.out.println("int: " + i);
    }
    public void printIt(double n) {
        System.out.println("double: " + n);
    }
}
```

```java
class Example {
    public static void main(String[] args) {
        A a = new A();
        a.printIt(3);
        a.printIt(3.0);
    }
}
```
### Review: Ancestors & Descendants

Polygon is a **descendant** of ClosedFigure

Polygon is an **ancestor** of Pentagon

Note: these are *not actually* Java built-in classes (this is just an example)

### Type Conformance

Objects conform to the types of all of their **ancestors**

Pentagon conforms to the types of both Polygon and ClosedFigure

### Using Conformance Properly

- Any sub-class that conforms to another class can be used anywhere something of the original type can be used
- For example, if we have this method declaration:

  ```java
  private void methodName( type1 var1, type2 var2 ) {...}
  ```

  And then use it:

  ```java
  methodName( object1, object2 );
  ```

### Using Conformance Properly (2)

- Typically, we have used **strict** conformance in our code
- For example, if we have this method declaration:

  ```java
  public void makeString( char a, char b, char c, char d ) {...}
  ```

  We must use it with the **same types** of inputs, since these are primitive types, with no conformance:

  ```java
  makeString( 'z', 'y', 'x', 'w' );
  ```

- Conformance allows us to relax this requirement when dealing with reference (Class) types, using **descendant** classes instead of the original
Using Inherited Conformance

- We can write methods using specific Oval sub-class, as usual:

  ```java
  public boolean isLarge(Oval o) {
    return o.getHeight() > 150;
  }
  ```

- Or we can write it on a more general class type:

  ```java
  public boolean isLarge(JComponent c) {
    return c.getHeight() > 150;
  }
  ```

- An advantage: this method will work for other types, like Rectangle, as well

  All such types inherit getHeight() from parent, JComponent

Using Inherited Conformance (2)

- Using a more general ancestor type like JComponent allows us to write methods in a more generic way

  This is in fact how some of the code we have been using all along actually works, like the Window class, with code like this:

  ```java
  public void add(JComponent component) {
    window.add(component, 0);
    component.repaint();
  }

  public void remove(JComponent component) {
    window.remove(component);
    window.repaint();
  }
  ```

  These methods work for any object that conforms to JComponent, like all the various buttons, text-boxes, shapes, etc. we have been using

  No need to re-write the method for each different shape or object

Over-Loading Uses Most Specific Types Possible

- If we over-load a method, using types that are in a conformance hierarchy, the compiler/JVM will use the one lowest (most specific) in the hierarchy

```java
class Example {
  public static void main(String[] args) {
    Driver driver = new Driver();
    Oval o = new Oval(...);
    driver.isLarge(o);
    Rectangle r = new Rectangle(...);
    driver.isLarge(r);
    RedDot dot = new RedDot(...);
    driver.isLarge(dot);
  }
}
```
Over-Loading Uses **Most Specific** Types Possible

- If we over-load a method, using types that are in a conformance hierarchy, the compiler/JVM will use the **one lowest (most specific)** in that hierarchy.

```java
class Example {
    public static void main(String[] args) {
        Driver driver = new Driver();
        Oval o = new Oval(...);
        driver.isLarge(o);
        Rectangle r = new Rectangle(...);
        driver.isLarge(r);
        RedDot dot = new RedDot(...);
        driver.isLarge(dot);
    }
}
```

- **Rectangle** does not conform to **Oval**, only to **JComponent**, so the **less specific** method version is used (it is the only version of the method that could work).

- **RedDot** conforms to both **Oval** and **JComponent**, so the **most specific** method version is used (while both methods would work, the **Oval** version actually gets called here).

---

**This Week & Next**

- **Meetings this week:**
  - Tuesday/Wednesday: regular classroom
  - Friday: in the CS Lab (16 Wing)
- **Reading 09**: Ch. 10 due Wednesday, 11 December, 12:00 PM
- **Program 08**: due Monday, 09 December, 11:59 PM
- **Office Hours This Week**: Wing 212
  - Monday/Friday: 2:15 PM–3:15 PM
  - Tuesday: 1:30–2:30 PM
  - Wednesday: 12:05–1:00 PM
- **Practice Final available soon**
- **SEIs now available. Please take a moment...**