
# Overriding Methods

# Override Occurs When…

- Parent class has a visible non-static method
  - A method which would normally be inherited by a child class
- Child class has the same method with the same parameters
  - Child class method overrides inherited parent method

- When a reference to the parent invokes the method, the parent method is invoked

- When a reference to the child invokes the method, the child method is invoked

# Virtual Method Table (VMT)

- Internal table computed at compile time by Java for each class
- Columns: 1- List of method names, 2 - Pointer to method code

| Shape VMT | |
|---|---|
| Method | Code |
| Shape(Point) | Shape.java:6 |
| move(double,double) | Shape.java:7 |
| min() | Shape.java:8 |
| toString() | Shape.java:9 |

overridden method

overriding method

| Rectangle VMT | |
|---|---|
| Method | Code |
| super(Point) | Shape.java:6 |
| Rectangle(Point,double,double) | Rectangle.java:8 |
| move(double,double) | Shape.java:7 |
| min() | Shape.java:8 |
| max() | Rectangle.java:14 |
| perimeter() | Rectangle.java:16 |
| area() | Rectangle.java:17 |
| toString() | Rectangle.java:19 |
| super.toString() | Shape.java:9 |

# Dynamic Dispatching

- When a non-static method is invoked:

1. Java run-time determines the dynamic type of the reference
2. Java looks up the method in the dynamic type VMT
3. Java invokes the code using the second column of the VMT

- Static methods do not use dynamic dispatch

# The @Override compiler annotation

- You may precede a method with @Override
  @Override public String toString() {

- If @Override is present, compiler will issue an error message if the method does **not** override some ancestor's method

- If @Override is **not** present, the method may or may not override some ancestor's method (no checking)

# The “final” keyword

- If a method is declared as **final**, sub-class CANNOT override!

```
public final boolean checkPassword(String pwd) {
        …
} // no-one can change this in a subclass
```

- If a class is declared as “final”, no sub-classes!

```
public final class String {
        …
} // cannot make a subclass
```

# The "super" keyword

- The "super" keyword refers to the VMT of the parent
  - You cannot use `super.`*`childMethod`*`()` or you get a compiler error

- The "super" keyword "affects" the dynamic type of "this"
  - If child "toString" overrides parent "toString", then `this.super.toString()` invokes the parent toString method!

- Note: Constructors cannot be overriden

# Private Methods

- A private method is not visible from outside the class
- If a private method is invoked from inside the class it will ALWAYS run the class method,
  - even if the reference has a dynamic sub-type that "overrides" that method!

- This makes private methods an exception to dynamic typing!
  - private methods do not respect the dynamic type of the reference
  - Even though the super-class may have a method with the same name that seemingly "overrides" the private method

# Example of Dynamic Dispatch

```
public class Parent {
        void who() { System.out.println("parent"); }
        public static void main(String[] args) {
                Child c = new Child(); // c static type = dynamic type = Child
                Parent p = c; // p static type = Parent, dynamic type = Child
                System.out.print("c.who() is: "); c.who();
                System.out.print("p.who() is: "); p.who();
        }
}
public class Child extends Parent {
        @Override public void who() { System.out.println("child"); }
}
```

Prints : "child"

Prints : "child"

@Override ok

# Example Private Method

```
public class Parent {
        private void who() { System.out.println(“parent”); }
        public static void main(String[] args) {
                Child c = new Child(); // c static type = dynamic type = Child
                Parent p = c; // p static type = Parent, dynamic type = Child
                System.out.print("c.who() is: "); c.who();
                System.out.print("p.who() is: "); p.who();
        }
}
public class Child extends Parent {
        public void who() { System.out.println(“child”); }
}
```

Prints : “child”

Prints : “parent”

@Override causes compiler error

# Default Method Visibility

- If you do not specify “private”, “public”, or “protected”, you get “package private” visibility
- “package private” methods behave like public methods within the package
- “package private” methods behave like private methods if invoked from outside the package… i.e. invisible
  - Also can violate dynamic typing… if **static** type is within package, and "override" is external to package, package private method within the package is invoked

# Protected Methods

- Behave like public methods if invoked within the package
- Visible within a descendant class, even if outside the package
  - And, you can only override from a descendant class
  - That means protected method respect dynamic typing!
- Invisible outside package from non-descendant classes!

# Method Visibility / Overridability

| Method Access | Overridable | Visible/Invocable |
|---|---|---|
| Private | No | Within class |
| "Package Private" | Within package | Within package |
| Protected | Every descendant class | Within package or descendant class |
| Public | Every descendant class | Everywhere |

# Method Invocation

1. For static method invocation, find method in the specified class.
2. If there is a private method referenced by the static type in this class/package, invoke it
3. Otherwise, Dynamic Dispatch…
    1. determine the dynamic type of the reference
    2. Look up the method in the dynamic type VMT
    3. Invoke the code using the second column of the VMT