**Background**

- OOMatch combines two areas of research: dispatch and pattern matching.
- Dispatch: which method is chosen for a call.
- Aside from regular dispatch in Java, two newer, more powerful forms are **Multimethods** and **Predicate Dispatch**.

**Multimethods**: Consider dynamic types of all arguments, and not just the class parameter.

```java
void draw(Shape s) {...}  //overrides //draw(Shape)
void draw(Circle c) {...}  //overrides //draw(Circle)
```

- The second draw overrides the first, because Circle is a subclass of Shape.
- In Java, the methods would merely be overloaded.

**Predicate Dispatch**: Can specify arbitrary predicates or preconditions to guard entry into a method.

```java
double log(double x) {...}  //overrides f(double)
when x <= 0 {...}  //overrides both methods
```

- When the predicate of one method m implies that of another method n, m overrides n.

**Pattern Matching**: Allows decomposition of an expression.

```java
match pair with
  (0, second) => second
  _ => ...
```

- This SML code checks whether “pair” is a tuple with 2 components, the first one being 0.
- “second” is a free variable that can be used later (in this case, it is returned).
- OOMatch allows pattern matching not just on built-in values but on Java objects.

**Introducing OOMatch**

- OOMatch allows parameters to be specified as patterns, in addition to simple variables.

```java
void f(Rect(Point(0, 0), Point p)) {}
```

- This function takes a single parameter of type Rect.
- It only applies if the Rect is composed of two points, and the first point has coordinates (0, 0).
- Any named variables in the pattern (p in this case) can be used in the method body.
- Patterns can be nested to any arbitrary depth.
- Methods with more specific patterns override those with less specific patterns:

```java
void f(Rect x) {}
void f(Rect(Point p1, Point p2)) {}
void f(Rect(Point(0, 0), Point(int x, int y))) {}  //overrides both methods
void f(Rect(ScreenCoord p1, ScreenCoord p2) r) {}  //overrides f(Rect)
void f(Rect(ScreenCoord(0, 0), ScreenCoord(int x, int y))) {}
```

- This results in the following override relationships between these methods:

```
(Rect)  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
</tr>
</tbody>
</table>
```

- For example, if a Rect with corners that are instances of ScreenCoord (a subclass of Point) were passed to f, the method `f(Rect(ScreenCoord p1, ScreenCoord p2) r)` would normally apply. Though `f(Rect(Point p1, Point p2))` is also applicable, it is less specific, because ScreenCoord is a particular type of Point.
- If the coordinates of the first ScreenCoord parameter were (0, 0), then the last method would apply instead, as it is the most specific.

**How is Pattern Matching on Objects Enabled?**

- Before doing pattern matching in method parameters, objects need **destructors**.
- The simple way to provide one:

```java
public class Rect {
    private Rect(private Point topLeft, private Point bottomRight){}
}
```

- The presence of access specifiers (private, protected and public are allowed) creates a **constructor** and **destructor** for class Rect all at once:
- Enables matching with the same parameters that were passed to the constructor.

**Explicit Destructors:**

- Used when more control is needed over what is considered the “components” of a class.
- Also enables private data to be kept private, while allowing matching:

```java
public class Rect {
    private Point topLeft, bottomRight;
    public Rect(Point topLeft, Point bottomRight) {
        this.topLeft = topLeft;
        this.bottomRight = bottomRight;
    }
    deconstructor Rect(Point topLeft, Point bottomRight) {
        topLeft = this.topLeft;
        bottomRight = this.bottomRight;
        return true;
    }
}
```

- Destructors are like regular methods: they can contain arbitrary code.
- Their parameters are **out parameters**, which the destructor **must** assign values to.
- The out parameters are matched against the pattern in method bodies.
- Return false from a destructor to explicitly prevent a match.