The **`Object`** Class

The `Object` class

- `Object` is the ultimate superclass of all Java classes, even the ones we designed

```
public class Circle { . . . }
```

is really

```
public class Circle extends Object { . . . }
```


`Object` class methods

- The `Object` class defines 11 methods
- We are most interested in two of the 11 methods

Method Name	Purpose
`equals`	Determines if two objects are to be considered "equal"
`toString`	Returns the string representation for an object

`toString` method

- `toString` is called when an object reference must be represented as a string
- `Point` class, no `toString` method:

```
pt = new Point(1, 2);
System.out.println(pt);
```

```
Point@34e265
```

- `Point` class, with `toString` method:

```
pt = new Point(1, 2);
System.out.println(pt);
```

```
(1,2)
```


`equals` method

- The equals operator (==) tests for *reference* equality
  - Do the object references point to the same object?
- Override `equals` method to define equality appropriately for your objects
- Caveat: parameter is an `Object` reference

Overriding `equals`

```
class Point {
    public boolean equals(Object other) {
        if (other == null) {
            return false;
        }
        if (other == this) {
            return true;
        }
        if (!(other instanceof MyClass)) {
            return false;
        }
        Point pt = (Point) other;
        return this.x == other.x && this.y == other.y;
}
```

Overriding `equals`

```
class Point {
    public boolean equals(Object other) {
        if (other == null) {
            return false;
        }
        if (other == this) {
            return true;
        }
        if (!(other instanceof MyClass)) {
            return false;
        }
        Point pt = (Point) other;
        return this.x == other.x && this.y == other.y;
}
```

Check for `null`

Check for reference equality

Cast and check type-specific characteristics

Check for correct equivalent type

Java's Type Dichotomy

- Primitive types vs. reference types
- Primitive type instances are not real objects in the OOP sense
- Primitive-type wrapper classes provide a bridge between the two kinds of objects
- Why primitive types at all?
  - Efficient
  - Map more directly to underlying hardware

Wrapper Classes for Primitive Types

Primitive Type	Wrapper Class
`boolean`	`Boolean`
`byte`	`Byte`
`char`	`Character`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

Boxing and Unboxing

- Assigning a primitive value to a wrapper variable
  - The primitive value is wrapped by the object
  - This is called *boxing*
- Assigning a wrapped primitive to a primitive variable
  - The wrapped value is extracted and assigned to the primitive variable
  - This is called *unboxing*
- The compiler boxes and unboxes automatically