

OO terminology

- *Object*: a structured collection of *fields* (aka *instance variables*) and *methods*
- *Class*: a template for creating objects
- The class of an object specifies...
 - the types and initial values of its local state (fields)
 - the set of operations that can be performed on the object (methods)
 - one or more *constructors*: code that is executed when the object is created (optional)
- Every (Java) object is an *instance* of some class

Objects in Java

```
public class Counter {
```

class name

```
private int r;
```

instance variable

```
public Counter () {  
    r = 0;  
}
```

constructor

```
public int inc () {  
    r = r + 1;  
    return r;  
}
```

methods

```
public int dec () {  
    r = r - 1;  
    return r;  
}
```

class declaration



object creation and use



```
public class Main {
```

```
public static void  
main (String[] args) {
```

constructor invocation

```
Counter c = new Counter();
```

```
System.out.println( c.inc() );
```

method call

```
}
```

Creating & Using Objects

- *Declare* a variable to hold a **Counter** object
 - Type of the object is the *name* of the class that creates it
- *Invoke* the *constructor* for **Counter** to create a **Counter** instance with keyword "new" and store it in the variable

```
Counter c = new Counter();
```

- *Invoke* the *methods* of an object instance using "dot"

```
c.inc();
```

What is the value of ans at the end of this program?

```
Counter x = new Counter();  
x.inc();  
int ans = x.inc();
```

1. 1
2. 2
3. 3
4. NullPointerException

Answer: 2

```
public class Counter {  
  
    private int r;  
  
    public Counter () {  
        r = 0;  
    }  
  
    public int inc () {  
        r = r + 1;  
        return r;  
    }  
  
}
```

What is the value of ans at the end of this program?

```
Counter x;  
x.inc();  
int ans = x.inc();
```

1. 1
2. 2
3. 3
4. NullPointerException

Answer: NPE

```
public class Counter {  
  
    private int r;  
  
    public Counter () {  
        r = 0;  
    }  
  
    public int inc () {  
        r = r + 1;  
        return r;  
    }  
  
}
```

What is the value of ans at the end of this program?

```
Counter x = new Counter();  
x.inc();  
Counter y = x;  
y.inc();  
int ans = x.inc();
```

1. 1
2. 2
3. 3
4. NullPointerException

```
public class Counter {  
    private int r;  
  
    public Counter () {  
        r = 0;  
    }  
  
    public int inc () {  
        r = r + 1;  
        return r;  
    }  
}
```

Answer: 3 x and y are *aliases*

Constructors with Parameters

```
public class Counter {  
  
    private int r;  
  
    public Counter (int r0) {  
        r = r0;  
    }  
  
    public int inc () {  
        r = r + 1;  
        return r;  
    }  
  
    public int dec () {  
        r = r - 1;  
        return r;  
    }  
}
```

Constructor methods can take parameters

Constructor must have the same name as the class

object creation and use

```
public class Main {  
  
    public static void main (String[] args) {  
        Counter c = new Counter(3);  
        System.out.println( c.inc() );  
    }  
}
```

Mutability

- Every Java variable is mutable

```
Counter c = new Counter(2);  
c = new Counter(4);
```

- A Java variable of *reference* type can also contains the special value “null”

```
Counter c = null;
```

Note:

Single = for assignment

Double == for reference equality testing

Null

- At any time, a Java variable of reference type can contain either “null” or a pointer into the heap
 - i.e., a Java variable of reference type "T" is like an OCaml variable of type "T option ref"
 - The dereferencing of the pointer and the check for “null” are implicitly performed every time a variable is used

```
let f (co : counter option ref) : int =  
  begin match co.contents with  
  | None ->  
    failwith "NullPointerException"  
  | Some c ->  
    c.inc()  
  end
```

```
class Foo {  
  public int f (Counter c) {  
    return c.inc();  
  }  
}
```

- If null value is used as an object (i.e. with a method call) then a NullPointerException occurs

Encapsulating local state

```
public class Counter {
```

```
    private int r;
```

```
    public Counter () {  
        r = 0;  
    }
```

```
    public int inc () {  
        r = r + 1;  
        return r;  
    }
```

```
    public int dec () {  
        r = r - 1;  
        return r;  
    }  
}
```

r is *private*

constructor and
methods can
refer to r

```
public class Main {
```

```
    public static void  
    main (String[] args) {
```

```
        Counter c = new Counter();
```

```
        System.out.println( c.inc() );
```

```
    }  
}
```

other parts of the
program can only access
public members

method call

Encapsulating local state

- Visibility modifiers make the state local by controlling access
- Basically:
 - `public` : accessible from anywhere in the program
 - `private` : only accessible inside the class
- Design pattern — first cut:
 - Make *all* fields private
 - Make constructors and non-helper methods public

(There are a couple of other protection levels — protected and “package protected”. The details are not important at this point.)

Java Core Language

differences between OCaml and Java

Expressions vs. Statements

- OCaml is an *expression language*
 - Every program phrase is an expression (and returns a value)
 - The special value () of type `unit` is used as the result of expressions that are evaluated only for their side effects
 - Semicolon is an *operator* that combines two expressions (where the left-hand one returns type `unit`)
- Java is a *statement language*
 - Two-sorts of program phrases: expressions (which compute values) and statements (which don't)
 - Statements are *terminated* by semicolons
 - Any expression can be used as a statement (but not vice-versa)

Types

- As in OCaml, every Java *expression* has a type
- The type describes the value that an expression computes

Expression form	Example	Type
Variable reference	x	Declared type of variable
Object creation	new Counter ()	Class of the object
Method call	c.inc()	Return type of method
Equality test	x == y	boolean
Assignment	x = 5	<i>don't use as an expression!!</i>

Type System Organization

	OCaml	Java
<i>primitive types</i> (values stored “directly” in the stack)	int, float, char, bool, ...	int, float, double, char, boolean, ...
structured types (a.k.a. <i>reference</i> <i>types</i> — values stored in the heap)	tuples, datatypes, records, functions, arrays <i>(objects encoded as records of functions)</i>	objects, arrays <i>(records, tuples, datatypes, strings, first-class functions are a special case of objects)</i>
<i>generics</i>	‘a list	List<A>
<i>abstract types</i>	module types (signatures)	interfaces public/private modifiers

Arithmetic & Logical Operators

OCaml	Java	
=, ==	==	equality test
<>, !=	!=	inequality
>, >=, <, <=	>, >=, <, <=	comparisons
+	+	addition (and string concatenation)
-	-	subtraction (and unary minus)
*	*	multiplication
/	/	division
mod	%	remainder (modulus)
not	!	logical "not"
&&	&&	logical "and" (short-circuiting)
		logical "or" (short-circuiting)

New: Operator Overloading

- The meaning of an operator is determined by the *types* of the values it operates on
 - Integer division
 $4/3 \Rightarrow 1$
 - Floating point division
 $4.0/3.0 \Rightarrow 1.3333333333333333$
 - Automatic conversion
 $4/3.0 \Rightarrow 1.3333333333333333$
- Overloading is a general mechanism in Java
 - we'll see more of it later

Equality

- like OCaml, Java has two ways of testing reference types for equality:


- “pointer equality”

- `o1 == o2`

- “deep equality”

- `o1.equals(o2)`

every object provides an “equals” method that “does the right thing” depending on the class of the object



- Normally, you should use `==` to compare primitive types and `“.equals”` to compare objects

Strings

- `String` is a *built in* Java class
- Strings are sequences of characters
 `"Mount Fuji"` `"3 Stooges"` `"富士山"`
- `+` means String concatenation (overloaded)
 `"3" + " " + "Stooges" ⇒ "3 Stooges"`
- Text in a String is immutable (like OCaml)
 - but variables that store strings are not
 - `String x = "OCaml";`
 - `String y = x;`
 - Can't do anything to `x` so that `y` changes
- **The `.equals` method returns true when two strings contain the same sequence of characters**

What is the value of ans at the end of this program?

```
String x = "CIS 120";  
String z = "CIS 120";  
boolean ans = x.equals(z);
```

1. true
2. false
3. NullPointerException

Answer: true

This is the preferred method of comparing strings.

What is the value of ans at the end of this program?

```
String x1 = "CIS ";  
String x2 = "120";  
String x = x1 + x2;  
String z = "CIS 120";  
boolean ans = (x == z);
```

1. true
2. false
3. NullPointerException

Answer: false

Even though x and z both contain the characters "CIS 120", they are stored in two different locations in the heap.

What is the value of ans at the end of this program?

```
String x = "CIS 120";  
String z = "CIS 120";  
boolean ans = (x == z);
```

1. true
2. false
3. NullPointerException

Answer: true(!)

Why? Because strings are immutable, two identical strings that are known when the program is compiled can be aliased.

Moral

Always use `s1.equals(s2)` to
compare strings!

You almost always want to compare strings with respect to their content, not where they are allocated in memory...

(But be warned: `s1` might be null!)

Style: naming conventions

Kind	Part-of-speech	Example
class	noun	RacingCar
(mutable) field, variable	noun	initialSpeed
(immutable) field, variable	noun	MILES_PER_GALLON
method	verb	shiftGear

- Identifiers consist of alphanumeric characters and `_` and cannot start with a digit
- The larger the scope, the more *informative* the name should be
- Conventions are important: variables, methods and classes can have the same name