Queens' College Cambridge

Object Oriented Programming

Alistair O'Brien

Department of Computer Science

January 12, 2021

Contents

1 Types, Objects and Classes

- **Declarative Language**: A programming paradigm that specifies what is to be done, not how to do it (e.g. SQL)
- **Imperative Language**: A programming paradigm that specifies both.
- **Statement**: A statement produces a side effect and doesn't have a value.
- **Expression**: An expression has a value (and may have side effects).
- **Primitive**: A fundamental built-in datatype
  e.g. `int, byte, short, long, float, double, boolean, char`

1.1 Classes and Objects

- **Class**: Template for creating objects.
- **Object**: Instance of a class. Created using a **constructor**.
- **Constructor**: Special method in the class used to create an object. Has no explicit return type and has same name as class name in Java. **Default constructor** zeros uninitialized class fields.
- **Type**: an interface.
- A function **prototype** specifies it's name, argument and return type (similar to function type in OCaml).
- Classes have a **modifier** and a **name**:
  - **Class State**: properties / variables / fields
  - **Class Behavior**: methods

1.2 Encapsulation

- **Encapsulation**: Concept of grouping internal state and methods that operate on the state, or restricting the direct access of state.

- Protective shield, prevents data being accessed by code outside the shield.
- Achieved using **access modifiers**.
- Accessor (getters) and mutators (setters) return / modify the state respectively.
- Promotes readability since we can explicitly see what parts of the internal state are being updated / used.

1.3 Modifiers

- **Static**:
  - Static fields are tied to class, not the instances. Behave rather like global variables.
  - Static methods are tied to class. Can be invoked without instantiating the class.
- **Final**: Allows immutability. Often indicates resource cannot be modified.

1.3.1 Access Modifiers

- **Private**: Accessible within the class.
- **Package**: Accessible within the same package. (default access)
- **Protected**: Accessible within subclasses.
- **Public**: Accessible anywhere.

2 Pointers, References and Memory

2.1 Memory

- Memory grouped into words (32-bit arch has 4 byte words).
- An address specifies a specific word

- **Call stack** grows down from 0. **Heap** grows up from `MAX_ADDR`.
- When a function is called, allocate new **stack frame**, contains local variables, etc.
- In C:
  - Stack: items exist for duration of function call, pointers used to access items on heap.
  - Items placed in heap until they are deleted (manually).
- In Java:
  - All primitive types go on stack.
  - Everything else goes on the heap. (e.g. arrays, strings, etc)
  - **Garbage Collector**: Automatically manages the heap.

2.2 References vs Pointers

- Java uses references, C uses pointers.
- Advantages:
  - Java references don't support arithmetic, whereas C pointers do. Allowing pointers to point to arbitrary addresses $\implies$ Exploits.
  - References only point to a single object $\implies$ cannot be assigned to arbitrary memory.
  - References are strongly typed. C pointers can be cast e.g. `int* -> char*`.
  - Dereferencing is automatic.
- References can be `null`, used to denote unassigned reference.
- Java uses **pass-by-value** when methods are called $\implies$ method arguments have their value **copied**. However, if object is passed, it is the **reference** that is copied, not the object. So deepcopy is required sometimes.

3 Lifecycle of an Object

3.1 Object Initialization

- **Static** fields are initialized when a the class is loaded into memory. They are not initialized when a variable for the class is declared.
- Static initializer block using `static { ... }`.
- Static fields are only initialized once.
- Instance fields are initialized when an object is instantiated.
- Object initialization order:
  1. Static fields in order of file
  2. Instance fields in order of file
  3. Constructor
- Initializing object of a subclass:
  1. `super.static`
  2. `subclass.static`
  3. `super.instance` and `constructor`
  4. `subclass.instance` and `constructor`

  This is known as **chaining**. If superclass constructor requires arguments, `super(args)` must be called explicitly.
- **Dangerous**: Bad practice to call methods from superclass constructor since overloaded methods may rely on uninitialized instance fields of the subclass.

3.2 Garbage Collection

3.2.1 Deterministic Destruction

- **RAII**: Resource Acquisition Is Initialization. Resources are acquired when object is instantiated. Resource must be freed when object is deleted (C/C++).

- Typically done using *destructor* (C/C++).
- In Java, the garbage collector deletes objects. Since programmers often forget to delete objects ($\implies$ memory leak) or try delete the same object multiple times ($\implies$ crash).
- Java uses **Try with Resources** (TWR).

```
try (Networker n = new Networker()) {
    // Object exists
}
// Object.close() automatically called, but object still exists

class Networker implements Closeable {
    ...
}
```

3.2.2 Garbage Collection

- Garbage collection is a separate background process monitoring the execution of the program.
- Garbage collection methods:
  - **Stop the World**: Pause the execution of the program while performing garbage collection.
  - **Incremental**: Collect in multiple phases and continue running the program in between (time sharing)
  - **Concurrent**: No pauses. Difficult to implement.
- Java uses a variant of "stop the world".
- Memory leaks can still occur e.g.

```
List<Leak> l = new ArrayList<>();
for (int i = 0; i < 10000000000; i++) l.add(new Leak());
```

- **Mark and Sweep**:

  - Start with the list of all accessible references.
  - Recursively follow each reference, marking each object.
  - Sweep the heap and remove unmarked objects.

4 Inheritance

- **Superclass**: The class being inherited from.
- **Subclass**: The class that inherits from another class.
- `extends` keyword is used.
- `final` classes cannot be extended
- `final` methods cannot be overridden.
- Subclass inherit properties from superclass:
  - **Code inheritance**: All protected fields and methods from superclass are available in subclass.
  - **Type inheritance**: Subclass can be used as an instance of superclass (Substitution principle)

4.1 Substitution Principle

- **Substitution Principle**: If $B \leq A$, then objects of type A can be replaced with objects of type B.

4.2 Casting

1. Java allows implicit *widening*, but *narrowing* must be explicit.
2. Can always cast up to a parent class
3. Can only cast to child if underlying object is child, otherwise run-time error
4. **Boxing**: Convert primitive type to it's `Object` counterpart.
5. **Unboxing**: Convert `Object` counterpart back to primitive type.

6. Java implicitly performs boxing and unboxing.

4.3 Overriding vs Overloading

- **Overloading**: Multiple methods with the same name but different arguments (and potentially different return types).
- Multiple methods cannot have the same name but different return types only.
- **Overriding**: Redefining behavior of an inherited method in the subclass.

4.4 Shadowing

- **Shadowing**: Occurs when variable declared in a scope has the same name as variable in outer scope.
- `this` is used to refer to current class, `super` is used to refer to parent class.

4.5 Multiple Inheritance

- Multiple inheritance is where a subclass can have one or more super classes.
- **Problem**: Occurs when a method in both super-classes have the same signature $\implies$ Compiler cannot determine method.
- **Diamond Problem**: Super-classes have a common super-class $\implies$ two copies of the same superclass? or a single one?
- Java doesn't allow multiple inheritance for these reasons. Instead uses **Interfaces**

4.6 Abstract Classes and Interfaces

Abstract Classes

- **Abstract Class**: A class that contains at least one `abstract` method, a method with no implementation.

- Cannot be instantiated, instead it is inherited by a (non-abstract) subclass and the abstract methods are implemented using overriding. Or can be inherited by another abstract class.

Uses of abstract classes

- Used to achieve abstraction, while providing default state and behavior for subclasses.
- Allows for more code reusability than a interface

Interfaces

- **Interfaces**: are used to specify behavior that a class must implement.
- Interfaces don't contain any concrete methods (or state).
- May contain `default` concrete methods and `public static final` variables (class constants).
- In case of conflicting `default` implementations in subclass, must explicitly provide implementation.

Uses of interfaces:

- Used to achieve abstraction
- Designed to support dynamic polymorphism
- Allows for separate definition of method from the Interface hierarchy
- Solves multiple inheritance problems.

4.7 Common Interfaces

- `<<interface>> Set`:
  - Collection of elements with no duplicates.
  - Implementations:
    - `TreeSet`: Elements stored in BST. Elements maintained in some total ordering.
    - `HashSet`: No defined ordering, but fast to operate on.

    * LinkedHashSet: Fast, ands retains ordering.

- <<interface>> List:
  - Ordered collection with duplicates.
  - Implementations:
    * LinkedList: Doubly-linked list data-structure
    * ArrayList: Resizable array.
    * Vector: Like ArrayList but is synchronized
- <<interface>> Map:
  - Used for dictionaries.
  - Implementations:
    * TreeMap: Elements stored in BST. Keys must have total ordering.
    * HashMap: No defined ordering on keys.
- <<interface>> Iterator:
  - Used to iterate over a collection. Obtained via collection.iterator().
  - Has .hasNext() and .next() methods.
  - Iterators are **fail fast**, they throw a ConcurrentModificationException if the underlying datastructure is modified during iteration.
  - Use .remove() to safely remove element returned by previous .next().

4.8 Covariance and Contravariance

- Ideas follow from substitution principle.
- **Covariant**: Preserves the ordering of types $\leq$.
- **Contravariant**: Reverses the ordering of types.

Covariance (Return values)

- If $B \leq A$, then $A.fun() \to X$ is substituted for $B.fun() \to Y$. Hence $Y \leq X$.

- We move **down** the inheritance hierarchy.

Contravariance (Parameters)

- If $B \leq A$, then $A.fun(X)$ is substituted for $B.fun(Y)$ where $X \leq Y$.
- We move **up** the inheritance hierarchy.
- **Not** supported by Java since if two method prototypes have different types but have the same name, then they are overloaded.
- **Arrays are Covariant**:
  - If $B \leq A$, then $B[] \leq A[]$.
  - Can lead to runtime errors e.g.

```
Integer[] i = new Integer[] { 1, 2 };
Object[] o = i;
o[0] = "Hello"; // Runtime error
```

- **Generics aren't Covariant**:
  - If $B \leq A$, then $\texttt{T<B>} \not\leq \texttt{T<A>}$.
  - Because non-contravariant parameters.

```
List<B> xs = List.of(...);
List<A> ys = xs; // Type safe, but not allowed. Compiler error

ys.set(0, new A()); // Not type safe.
```

  - Wildcard are used to fix this.

5 Polymorphism

5.1 Static Polymorphism

- Decided at compile time
- Since we don't know what the true type of object, parent type is used.

- Type errors give compiler errors
- Static Polymorphism applied to fields.

5.2 Dynamic Polymorphism

- **All methods are dynamic polymorphic**.
- Run the method of the child
- Must be done at run-time since that is when we know it's type.
- Type errors cause run-time errors.
- Dynamic Polymorphism has overhead $\implies$ static polymorphism is more efficient.

6 Generics

- Generic types implemented using type parameters e.g. `T`.
- Compiler uses **type-erasure**: replaces `T` with `Object` class, and then adds type-casting for type safety.

6.1 Wildcards

- `?` is used to denote a wildcard.
- Suppose $C \leq B \leq A$.

```
List<B> lb = new ArrayList<>();
List<A> la = lb; // Compiler error
```

- Generics aren't covariant, so we have a compiler error.
- Wildcards are the solution.

```
List<?> lA = new ArrayList<A>(); // unknown wildcard
List<? extends A> lE = new ArrayList<A>(); // extends wildcard
List<? super A> lS = new ArrayList<A>(); // super wildcard
```

- **Unknown Wildcard**:
  - `List<?>` means a list typed to an unknown type.
  - Could be `A, B, String, Object`, etc
  - Can only read `Object`s from collection.
  - Cannot write.

```
public void foo(List<?> l) {
    for (Object e : l) {
        System.out.println(e);
    }
}
```

- **Extends Wildcard**:
  - `List<? extends A>` means a `List` of objects that are instances of the class `A`, or subclasses of `A` (e.g. `B, C`).
  - Can read `A` (or subtypes `B, C`) objects from collection.
  - Cannot write.

```
public void foo(List<? extends A> l) {
    for (A e : l) {
        e.doStuff();
    }
}
```

- **Super Wildcard**:
  - `List<? super A>` means that the list is typed to either the `A` class or a superclass of `A`.
  - Can write instances of `A` (or subtypes `B, C` by substitution principle).

- Cannot read instances of type `A`. Only possible to read `Objects`.

```
public void insert(List<? super A> l) {
    l.add(new A());
    l.add(new B());
    l.add(new C());
}
```

7 Comparisons

7.1 Reference Equality

- **Reference Equality**: `ref == ref'`.
- Checks whether they point to the same memory address.

7.2 Structural Equality

- **Structural Equality**: `obj1.equals(obj2)`.
- By default, performs reference equality.
- Features of `equals` method:
  - **Reflexive**: $\forall x.$ `x.equals(x) == true`
  - **Symmetric**: $\forall x, y.$ `x.equals(y) == y.equals(x)`
  - **Transitive**: $\forall x, y, z.$ `x.equals(y) && y.equals(z) => x.equals(z)`
  - $\forall x.$ `x.equals(null) == false`.
  - **Consistent**: `equals` is a "pure function", always returning the same result.
- When overriding `equals`, also override `int hashCode()`

7.3 Comparable Interface

- Implements the `Comparable<T>` interface and overrides the `compareTo` method.

- Returns an integer: e.g. `this.compareTo(obj)`

$$
\begin{aligned}
r < 0 &\iff \text{this} < \text{obj} \\
r = 0 &\iff \text{this} = \text{obj} \\
r > 0 &\iff \text{this} > \text{obj}
\end{aligned}
$$

- Defines a total ordering over the type.
- Features of `compareTo` method:
  - **Antisymmetric**: $\forall x, y.$ `x.compareTo(y) <= 0 && y.compareTo(x) <= 0 => x.equals(y) == true`
  - **Transitive**: $\forall x, y, z.$ `x.compareTo(y) <= 0 && y.compareTo(z) <= 0 => x.compareTo(z) <= 0`

7.4 Comparator Interface

- Make a separate class which implements `Comparator<T>` $\implies$ a single type `T` can have multiple comparators.
- Override `compare(T x, T y)`
- Returns an integer. Same behavior as `compareTo`.

8 Errors

8.1 Types of Errors

- **Syntactic Errors**: Errors in programs syntax
- **Logical Errors**: Errors in the logic of the program
- **External Errors**: Errors caused by external processes the program relies on.

8.2 Returned Codes

- Return value can be ignored.
- Have to keep checking what return values might signify
- The actual result often can't be returned in the same way
- Forces one value in the result range to denote an error, typically `-1`.
- Results in mixing of code and error handling.

8.3 Deferred Error Handling

- Set some state in the system that needs to be checked for errors.

8.4 Exceptions

- An object that can be thrown and caught
- Extends `Exception`.
- Use `try ... catch ... finally`. `finally` block is **always** executed.
- Once any catch block is matched, the rest will be skipped $\Longrightarrow$ design pattern: start with the most specific and then generalize.
- The exception object can contain state that gives detail on the error.
- Exceptions can be chained. e.g.

```
class SomeException extends Exception {
    public SomeException(String message, Throwable cause) {
        super (message, cause);
    }
}
try {
    // something
} catch (DivideByZeroException e) {
    throw new SomeException("error", e);
}
```

- Both `Exception` and `RuntimeException` extend `Throwable`.

8.5 Checked Exceptions

- Extends `Exception`.
- Used for recoverable errors.
- Must be handled or passed up e.g. `int foo() throws IOException`

8.6 Unchecked Exceptions

- Extends `RuntimeException`
- Do have to be checked or declared.

8.7 Advantages and Disadvantages

- **Advantages**:
  - Exceptions can have descriptive names and hold lots of information
  - Can't be ignored by accident.
  - Avoids `Maybe` types.
- **Disadvantages**:
  - Can lead to surprising control flow.
  - Unrolls stack but doesn't rollback state $\implies$ partial results.
  - Less elegant for concurrent programs.

9 Class Taxonomy

- Java allowed *nested classes*.
- **Static Nested Classes**:
  - Nested class with `static` modifier:

```
public class OuterClass {
    static class StaticNestedClass { ... }
}
```

- Cannot refer to instance variables / methods.
- Instantiated using the following syntax

```
OuterClass.StaticNestedClass obj = new OuterClass.StaticNestedClass();
```

- **Inner Instance Classes**:
  - Non-static nested class:

```
public class OuterClass {
    class InnerInstanceClass { ... }
}
```

  - Associated with instance of `OuterClass`.
  - Direct access to the object's methods and state.
  - Cannot define any static methods / variables.
  - Instantiated using the following syntax:

```
OuterClass.InnerInstanceClass obj = outerObj.new InnerInstanceClass();
```

- **Local Classes**:
  - Defined in a block
  - Typically in methods. Also known as *method-local-classes*.
  - Scoping Rules:
    * Cannot be instantiated outside the block
    * Can access variables from enclosing block. But these variables are *effectively final*.
    * Has access to members of it's enclosing class.
  - Can implement a number of interfaces and (or) extend a class.

- **Anonymous Inner Classes**:
  - A local class that isn't bound to an identifier
  - Useful making an instance of a class with certain extras e.g. overloaded methods
  - Doesn't pollute the namespace
  - Syntax:

    ```
    AnonClass obj = new AnonClass() {
        @Override
        public void someMethod() { ... }
    };
    ```

  - Can only implement a single interface or extend a class.
  - Cannot contain a constructor.
  - Rules:
    * Has access to members of it's enclosing class.
    * Has access to effectively final variables in it's enclosing scope.
- **Lambda**:
  - **Functional Interface**: A functional interface is an interface that only contains a single abstract method. May contain any number of `default` methods.
  - Anonymous inner class syntax unwieldy, unclear and bulky for a single method implementation.
  - Often uses when passing functionality as an argument.
  - Lambda functions provide a solution:

    ```
    (args) -> expression or { ... };
    ```

10 Design Patterns

10.1 The Open-Close Principle

- Classes should be open for extension: Add additional functionality

- But closed to modification: extend the class without modifying it.

10.2 Types of Patterns

- **Creational Patterns**: Patterns concerned with the creation of objects (e.g. singleton)
- **Structural Patterns**: Patterns concerned with the compositions of classes / objects (e.g. decorator)
- **Behavioral Patterns**: Patterns concerned with how classes or object interact with and distribute resources (e.g. observer, state, strategy)

10.3 The Decorator Pattern

- How to add state / behavior at runtime?
- Wrap object in another object to dynamically change behavior.
- UML:

10.4 The Singleton Pattern

- How to ensure only one instance of an object is created?
- Recipe:
  - Make the constructor private
  - Create a single `static private` instance of the class inside the class
  - Create a `public static` getter method that returns the instance

e.g

```
class Singleton {
    private Resource resource;
    private Singleton() { resource = ...; }
    private static Singleton instance = new Singleton();
    public static Resource getInstance() { return resource; }
}
```

10.5 The State Pattern

- How can we let an object alter its behavior when it's internal state changes?
- Have some `Context` object that contains an instance of the `State` interface. Each behavior for state is implemented in some concrete subclass of `State`.
- UML:

10.6 The Strategy Pattern

- How can we select an algorithm implementation at runtime?
- Have some `Context` object that contains an instance of the `Strategy` interface. The different implementations of the algorithm are in the different concrete subclasses.
- UML:

10.7 The Composite Pattern

- How can we treat a group of objects as a single object?
- Have a `Component` interface. Leaves and composites inherit this:
  - Leaves implement the behavior.
  - Composites implement it by looping through it's children and then performing some optional extra behavior.
- UML:

10.8 The Observer Pattern

- When an object changes state, how can any interested parties know?
- The subject has a list of interested parties `observers`.
- When an event occurs, it iterates through `observers` and `notify`s them. e.g

```java
public class Subject {
    public interface Observer {
        void notify(Event e);
    }

    private State state = ...
    private List<Observer> observers = new ArrayList<>();

    public void attach(Observer o) {
        observers.add(o);
    }

    public void doSomething() {
        // do something
        for (Observer o : observers) {
            o.notify(Event e);
        }
    }
}
```