Today: **Design Patterns III**

**Detour: The “simplest” creational pattern: Singleton**

- Ensure a class only has one instance, and provide a global point of access to it
- In my experience, if there is one pattern people know of, it’s Singleton.
  - Catchy name
  - Simple concept
- Again, in my experience, if there is one pattern people get wrong, it’s Singleton.
  - Used as a simple substitute for global variables
  - ...while thinking that there is nothing wrong with that: “They’re not global variables – they’re a Singleton!”
- And even when people get the concept right, they frequently get the implementation wrong.
  - Threatened by parallel programs
  - Subclassing vs. protecting the one and only instance

**When is it OK to use Singletons?**

- Sometimes we need only one instance of an object
  - One file system
  - One print spooler
- If you think you need only one instance of the object, answer these questions:
  - Can you assign ownership of the single instance reasonably?
  - Is lazy initialization desirable?
  - Is global access not provided otherwise?
- If all criteria are satisfied, you might need a Singleton after all.
  - Threatened by parallel programs
  - Subclassing vs. protecting the one and only instance

**Singleton’s structure**

- Singleton
  - Defines a class-scoped instance() operation that lets clients access its unique instance
  - May be responsible for creating its own unique instance

```
Singleton

static Instance();
Singleton(operation);
GetSingleton();
Singleton.uniqueInstance
```
Sample code (not the real deal)

```java
public class SingletonObject {
    private SingletonObject() {
        // Just making the constructor private...
    }
    public static SingletonObject getSingletonObject() {
        if (ref == null) {
            // The object has not been created yet
            ref = new SingletonObject();
            return ref;
        }
        private static SingletonObject ref;
    }
}
```

But there’s a threading problem...

```java
public class SingletonObject {
    private SingletonObject() {
        // No code, just making the constructor private...
    }
    public static synchronized SingletonObject getSingletonObject() {
        if (ref == null) {
            // The object has not been created yet
            ref = new SingletonObject();
            return ref;
        }
        private static SingletonObject ref;
    }
}
```

Attack of the clones

```java
public class Clone {
    public static void main(String args[]) throws Exception {
        // Get a singleton
        SingletonObject obj = SingletonObject.getSingletonObject();
        // Oh crap...
        SingletonObject clone = (SingletonObject) obj.clone();
    }
}
```

- Note: Only really a problem if your Singleton class is extending another class that supports cloning
  - ...since clone() is a protected method

Sample code (the real deal)

```java
public class SingletonObject {
    private SingletonObject() {
        // Just making the constructor private...
    }
    public static synchronized SingletonObject getSingletonObject() {
        if (ref == null) {
            // The object has not been created yet
            ref = new SingletonObject();
            return ref;
        }
        public Object clone() throws CloneNotSupportedException {
            // You shall not pass!
            throw CloneNotSupportedException;
        }
        private static SingletonObject ref;
    }
}
```
Consequences

- **Controlled access to sole instance**
  - Because Singleton encapsulates the sole instance, it has strict control
- **Reduced namespace**
  - One access method only
- **Variable number of instances**
  - You could change your mind to have n (e.g. 5) instances

Implementation

- **Implementation is very language-dependent**
  - ...and Singletons are more necessary in some language than others
- **Not an excuse for using global variables!**
  - "The Singleton design pattern is one of the most inappropriately used patterns. Singletons are intended to be used when a class must have exactly one instance, no more, no less ... [Designers] frequently use Singletons in a misguided attempt to replace global variables ... A Singleton is, for intents and purposes, a global variable. The Singleton does not do away with the global; it merely renames it." – Jim Hyslop

Back to that pretty maze of ours...

- We’ve explored several ways to construct the elements of the maze
  - Factory methods
  - Abstract factories
  - Prototypes
- We’ll see one more and, with that, finish our tour through creational patterns...

Introducing the **Builder**

- **Separate the construction of a complex object from its representation so that the same construction process can create different representations**
  - E.g. read in Rich Text Format, converting to many different formats on load.
Applicability

- Use when:
  - The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled
  - The construction process must allow different representations for the object that is constructed

Structure

- Builder
  - Specifies an abstract interface for creating parts of a Product object
- Concrete Builder
  - Constructs and assembles parts of the product by implementing the Builder interface
  - Defines and keeps track of the representation it creates
  - Provides an interface for retrieving the product
- Director
  - Constructs an object using the Builder interface
- Product
  - Represents the complex object under construction
  - Includes classes that define the constituent parts, including interfaces for assembling the parts into the final result

Collaborations

- The Client creates the Director object and configures it with the Builder object
- Director notifies the Builder whenever a part of the product should be built
- Builder handles requests from the director and adds parts to the product
- The client retrieves the product from the Builder

Sample code

```java
public abstract class MazeBuilder {
    public void buildRoom(int r) {}
    public void buildDoor(int r1, int direction, int r2) {}
    public Maze getMaze() {return null;}
}

public class MazeGame {
    public Maze createMaze(MazeBuilder b) {
        b.buildRoom(1);
        b.buildRoom(2);
        b.buildDoor(1, Direction.North, 2);
        return b.getMaze();
    }
}
```
Sample code

```java
public class StandardMazeBuilder extends MazeBuilder {
  private Maze currentMaze;

  public Maze getMaze() {
    if (currentMaze == null) {
      currentMaze = new Maze();
    }
    return currentMaze;
  }

  public void buildRoom(int r) {
    if (getMaze().getRoom(r) == null) {
      Room room = new Room(r);
      getMaze().addRoom(room);
      for (int d = Direction.First; d <= Direction.Last; d++)
        room.setSide(d, new Wall());
    }
  }
}
```

Sample code

```java
public class StandardMazeBuilder extends MazeBuilder {
  private Maze currentMaze;

  public Maze getMaze() {
    if (currentMaze == null) {
      currentMaze = new Maze();
    }
    return currentMaze;
  }

  public void buildRoom(int r) {
    if (getMaze().getRoom(r) == null) {
      Room room = new Room(r);
      getMaze().addRoom(room);
      for (int d = Direction.First; d <= Direction.Last; d++)
        room.setSide(d, new Wall());
    }
  }
}
```

Sample code

```java
public class CountingMazeBuilder extends MazeBuilder {
  private int rooms = 0;
  private int doors = 0;

  public void buildDoor(int r1, int d, int r2) {
    doors++;
  }

  public void buildRoom(int r) {
    rooms++;
  }

  public int getDoors() { return doors; }
  public int getRooms() { return rooms; }
}
```

Sample code

```java
public class CountingMazeBuilder extends MazeBuilder {
  private int rooms = 0;
  private int doors = 0;

  public void buildDoor(int r1, int d, int r2) {
    doors++;
  }

  public void buildRoom(int r) {
    rooms++;
  }

  public int getDoors() { return doors; }
  public int getRooms() { return rooms; }
}
```

Sample code

```java
public class MazeGame {
  public static void main(String args[]) {
    MazeGame mg = new MazeGame();
    Maze m = mg.createMaze(new StandardMazeBuilder());
    System.out.println(m);
    CountingMazeBuilder cmb = new CountingMazeBuilder();
    mg.createMaze(cmb);
    System.out.println("rooms = \"+cmb.getRooms()+\"\);
    System.out.println("doors = \"+cmb.getDoors()+\"\);
  }
}
```
Sample code (Abstract factory reminder)

```java
public Maze createMaze(MazeFactory f) {
    Room r1 = f.makeRoom(1);
    Room r2 = f.makeRoom(2);
    Door d = f.makeDoor(r1, r2);
    r1.setSide(Direction.North, f.makeWall());
    r1.setSide(Direction.East, d);
    r1.setSide(Direction.West, f.makeWall());
    r1.setSide(Direction.South, f.makeWall());
    r2.setSide(Direction.North, f.makeWall());
    r2.setSide(Direction.East, f.makeWall());
    r2.setSide(Direction.West, d);
    r2.setSide(Direction.South, f.makeWall());
    Maze m = f.makeMaze();
    m.addRoom(r1);
    m.addRoom(r2);
    return m;
}
```

Sample code (Builder comparison)

```java
public Maze createMaze(MazeBuilder b) {
    b.buildDoor(1, Direction.North, 2); // A bit extreme, but you get the point...
    return b.getMaze();
}
```

Consequences

- Lets you vary a product’s internal representation
  - And hides details on how the product is assembled
- Isolates code for construction and representation
  - Clients don’t need to know anything about the classes that define the product’s internal structure; such classes don’t appear in Builder’s interface
- Gives you finer control over the production process
  - Constructs the product step by step under the director’s control, instead of in one shot

Implementation

- Assembly and construction interface
  - Builders construct their products in step-by-step fashion.
  - Builder class interface must be general enough to allow the construction of products for all kinds of concrete builders
- Why no abstract class for products?
  - In the common case, the products produced by the concrete builders differ so greatly in their representation that there is little to gain from giving different products a common parent class
Creational patterns recap

- If `createMaze()` calls virtuals to construct components
  - Factory method
- If `createMaze()` is passed a parameter object to create rooms, walls, and all other elements of the same family
  - Abstract factory
- If `createMaze()` is passed a parameter object to create and connect mazes step by step
  - Builder
- If `createMaze()` is parameterized with various prototypical rooms, doors, walls, ..., which it copies and then adds to the maze
  - Prototype
- If we need to ensure that there is only one maze per game, or one factory that produces its elements
  - Singleton