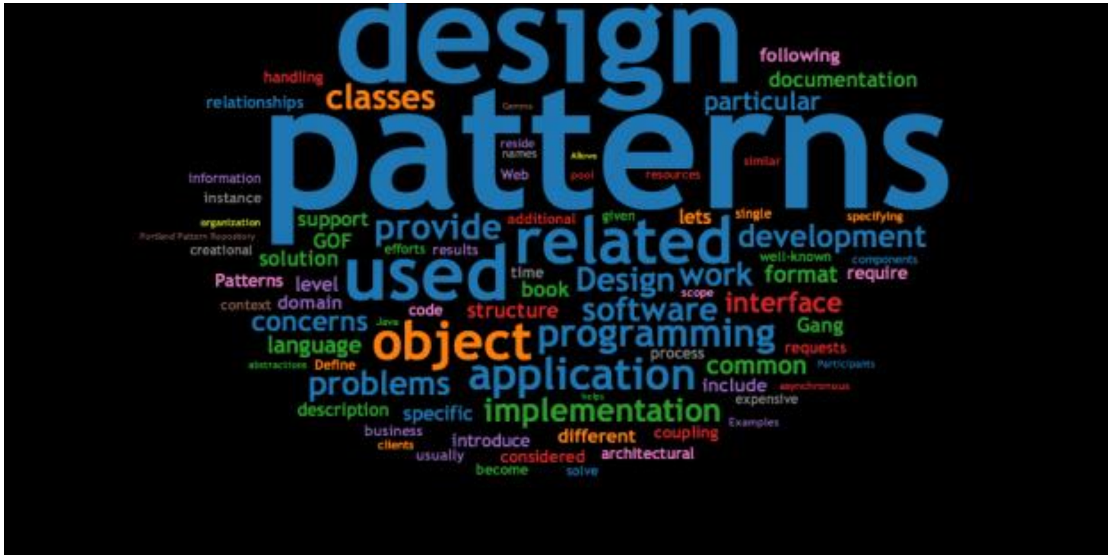


Design Patterns

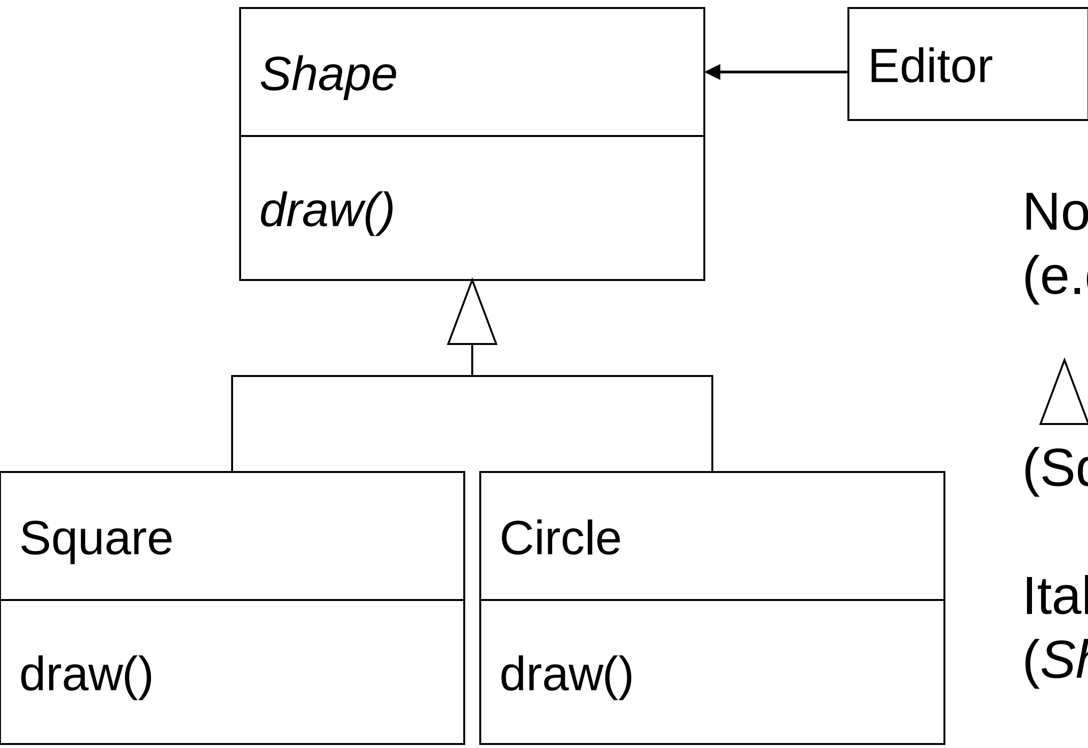
Outline of today’s class

- The Unified Modeling Language (UML)
- Design patterns
  - Intro to design patterns
  - Creational patterns
    - Factories: Factory method, Factory object, Prototype
    - Sharing: Singleton, Interning
  - Structural patterns
    - Adapter, Composite, Decorator, Proxy

UML Class Diagrams

- Unified Modeling Language (UML) is the “lingua franca” of object-oriented modeling and design
- UML class diagrams show classes, their interrelationships (inheritance and composition), their attributes and operations
- Also, UML sequence diagrams show dynamics of the system

Classes and Inheritance

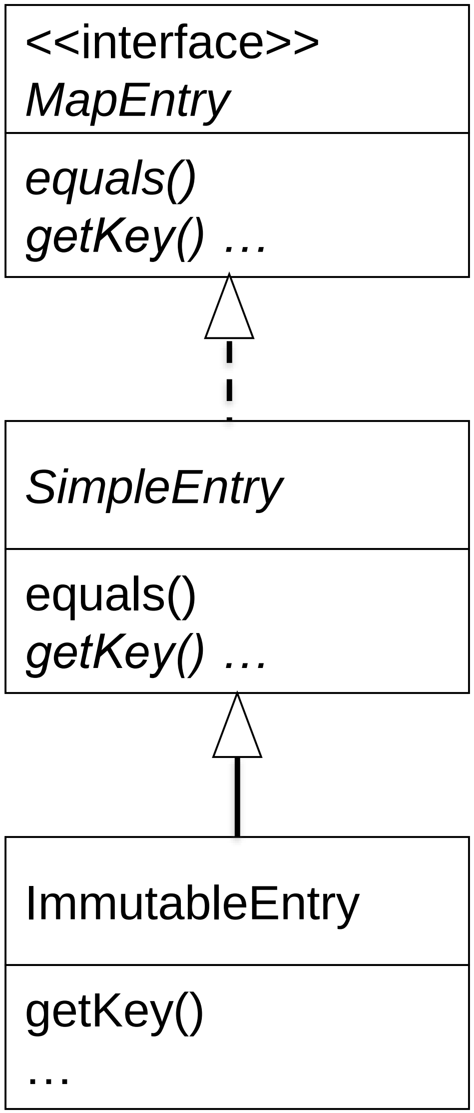


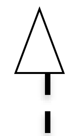
Notation: Boxes are classes (e.g., Shape, Circle).

denotes inheritance (Square is a subclass of Shape.)

Italics denote abstract (*Shape* is abstract, *draw()* is abstract)

Classes and Inheritance

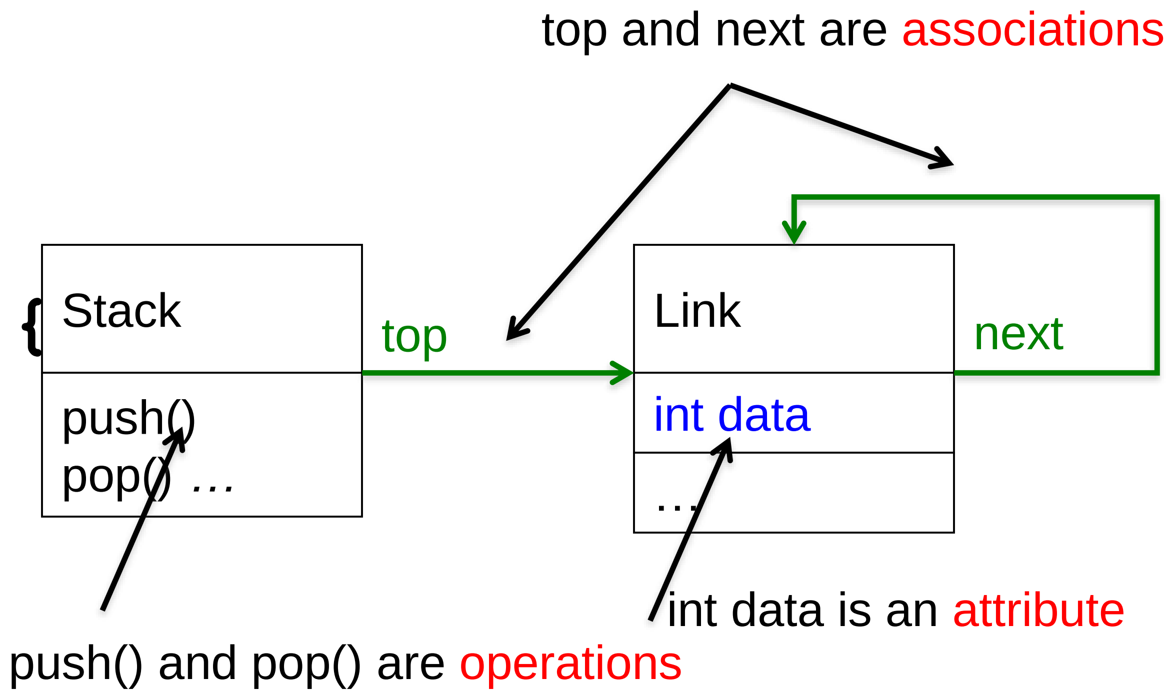


 denotes interface inheritance (*SimpleEntry* implements interface *MapEntry*)

ImmutableEntry extends abstract class *SimpleEntry*

Associations

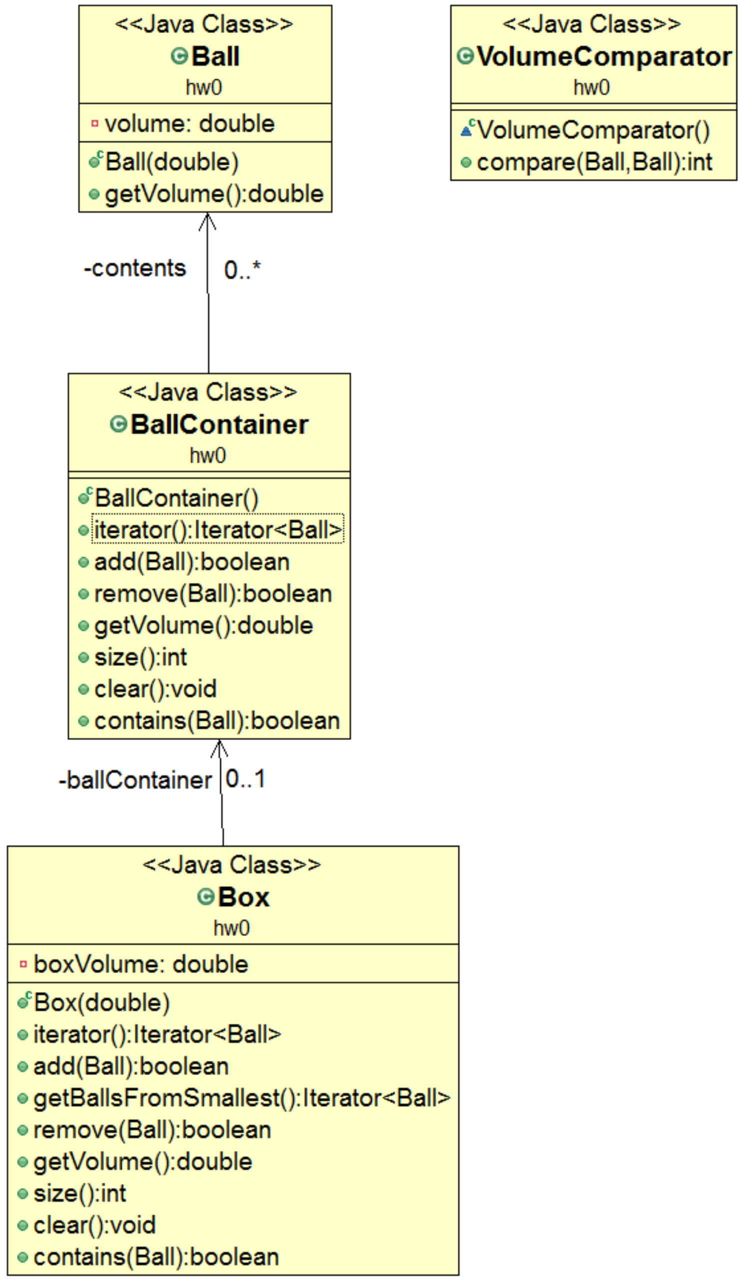
```
class Stack {
    Link top;
    ...
}
class Link {
    Link next;
    int data;
    ...
}
```

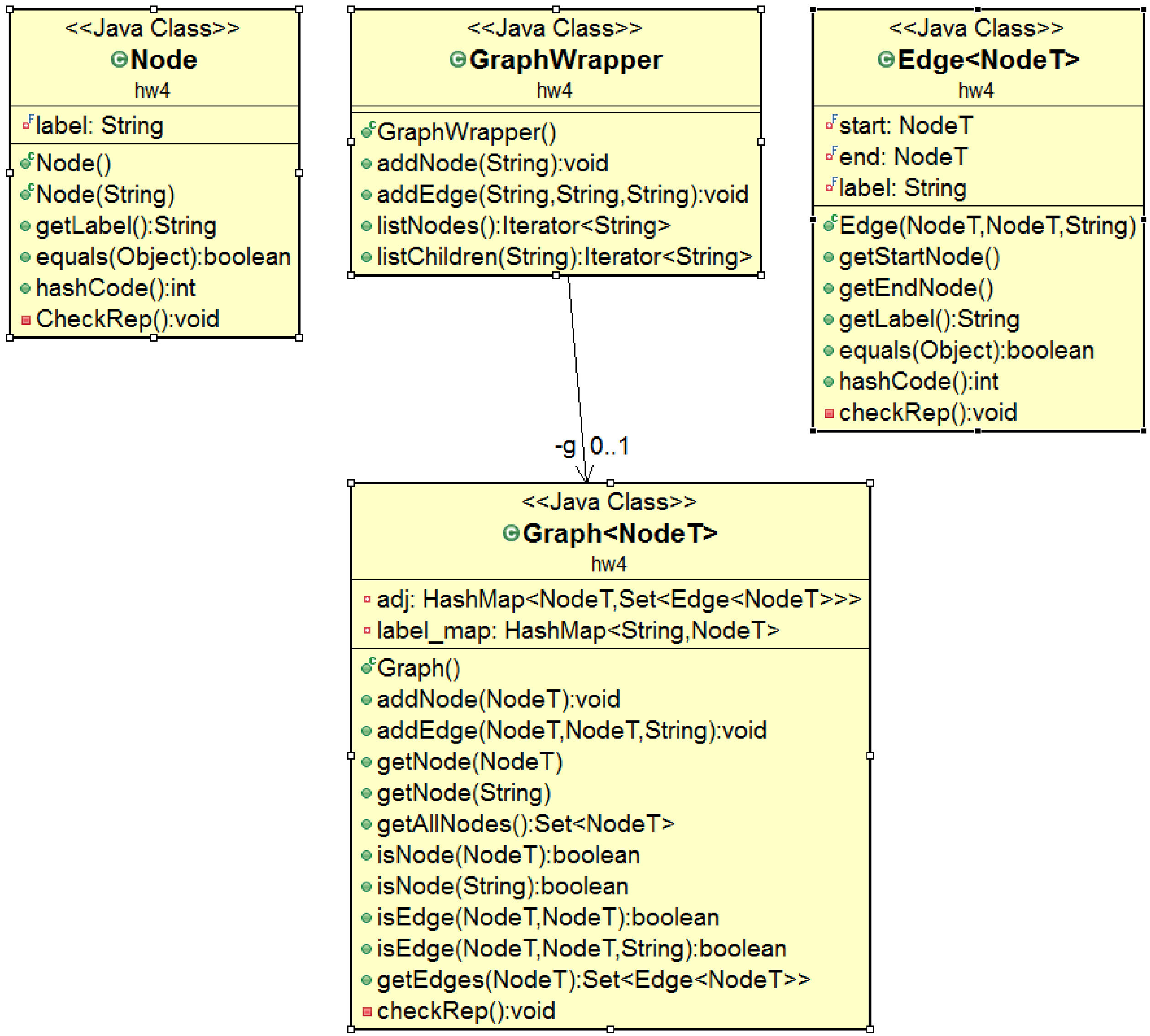


A UML association often represents a composition relationship: object of one class encloses the object(s) of the other. Above, Stack -> Link is a typical composition (has-a) relationship --- the Stack encloses/encapsulates its Link.

Exercise

- Draw a UML class diagram that shows the interrelationships between the classes from HW0


<<Java Class>>
Ball
hw0
volume: double
Ball(double)
getVolume():double
<<Java Class>>
VolumeComparator
hw0
VolumeComparator()
compare(Ball,Ball):int
-contents
0..*
<<Java Class>>
BallContainer
hw0
BallContainer()
iterator():Iterator<Ball>
add(Ball):boolean
remove(Ball):boolean
getVolume():double
size():int
clear():void
contains(Ball):boolean
-ballContainer
0..1
<<Java Class>>
Box
hw0
boxVolume: double
Box(double)
iterator():Iterator<Ball>
add(Ball):boolean
getBallsFromSmallest():Iterator<Ball>
remove(Ball):boolean
getVolume():double
size():int
clear():void
contains(Ball):boolean


<<Java Class>>
Node
hw4
label: String
Node()
Node(String)
getLabel():String
equals(Object):boolean
hashCode():int
CheckRep():void
<<Java Class>>
GraphWrapper
hw4
GraphWrapper()
addNode(String):void
addEdge(String,String,String):void
listNodes():Iterator<String>
listChildren(String):Iterator<String>
<<Java Class>>
Edge<NodeT>
hw4
start: NodeT
end: NodeT
label: String
Edge(NodeT,NodeT,String)
getStartNode()
getEndNode()
getLabel():String
equals(Object):boolean
hashCode():int
checkRep():void
-g
0..1
<<Java Class>>
Graph<NodeT>
hw4
adj: HashMap<NodeT,Set<Edge<NodeT>>>
label_map: HashMap<String,NodeT>
Graph()
addNode(NodeT):void
addEdge(NodeT,NodeT,String):void
getNode(NodeT)
getNode(String)
getAllNodes():Set<NodeT>
isNode(NodeT):boolean
isNode(String):boolean
isEdge(NodeT,NodeT):boolean
isEdge(NodeT,NodeT,String):boolean
getEdges(NodeT):Set<Edge<NodeT>>
checkRep():void

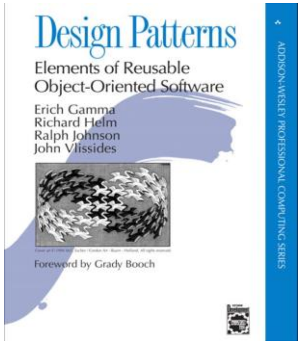
UML

- Can use UML to model abstract concepts (e.g., Meeting) and their interrelationships
  - Attributes and associations correspond to <u>specification fields</u>
  - Operations correspond to ADT operations
- Can use UML to express designs
  - Close correspondence to implementation
  - Attributes and associations correspond to <u>representation fields</u>
  - Operations correspond to <u>methods</u>

Design Patterns

- A design pattern is a <u>solution</u> to a design problem that occurs <u>over and over again</u>

- <u>The reference</u>: Gang of Four (GoF) book
  - "Design Patterns: Elements of Reusable Object-Oriented Software", by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (the Gang of Four), Addison Wesley 1995
  - Documents 23 still widely used design patterns



Design Patterns

- Design patterns promote extensibility and reuse
  - Help build software that is open to extension but closed to modification
    - the "Open/Closed principle"

- (Majority of) design patterns exploit subtype polymorphism

Why Should You Care?

- You can discover those solutions on your own
  - But you shouldn't have to
- A design pattern is a known solution to a known problem
  - Well-thought software uses design patterns extensively
  - Understanding software requires knowledge of design patterns

Design Patterns Don't Solve All Problems

- But, they can help
  - Get something basic working first
  - Improve it once you understand it
- Design patterns can increase or decrease understandability
  - Improve modularity, separate concerns, ease description
  - Add indirection, increase code size
- If your design or implementation has a problem, consider design patterns that address that problem
  - Canonical reference: the "Gang of Four" book
    - *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995.
  - Another good reference for Java
    - *Effective Java: Programming Language Guide* by Joshua Bloch, Addison-Wesley, 2001.

Design Patterns

- Three categories
  - Creational patterns (deal with object creation)
  - Structural patterns (control object structure, also known as heap layout)
  - Behavioral patterns (control object behavior)

Creational Patterns

- Problem: constructors in Java (and other OO languages) are inflexible
  1. Can't return a subtype of the type they belong to
  2. Always return a fresh new object, can't reuse
- "Factory" creational patterns present a solution to the first problem
  - Factory method, Factory object, Prototype
- "Sharing" creational patterns present a solution to the second problem
  - Singleton, Interning

Factories

- Problem: client desires more control over object creation
- Factory Method
  - Hides decisions about object creation
  - Implementation: put code in methods in client
- Factory object
  - Bundles factory methods for a family of types
  - Implementation: put code in a separate object
- Prototype
  - Every object is a factory, can create more objects like itself
  - Implementation: put code in clone methods

Motivation for Factories

- Supertypes support multiple implementations
  - Interface Matrix { ... }
  - class SparseMatrix implements Matrix { ... }
  - class DenseMatrix implements Matrix { … }
- Clients use the supertype (Matrix)
  - Still need to use a SparseMatrix or DenseMatrix constructor
  - Switching implementations requires code changes

Factory Instead

- Factory
- Clients call createMatrix, not a particular constructor
- Advantges
  - To switch the implementation, only change one place
  - Can decide what type of matrix to create

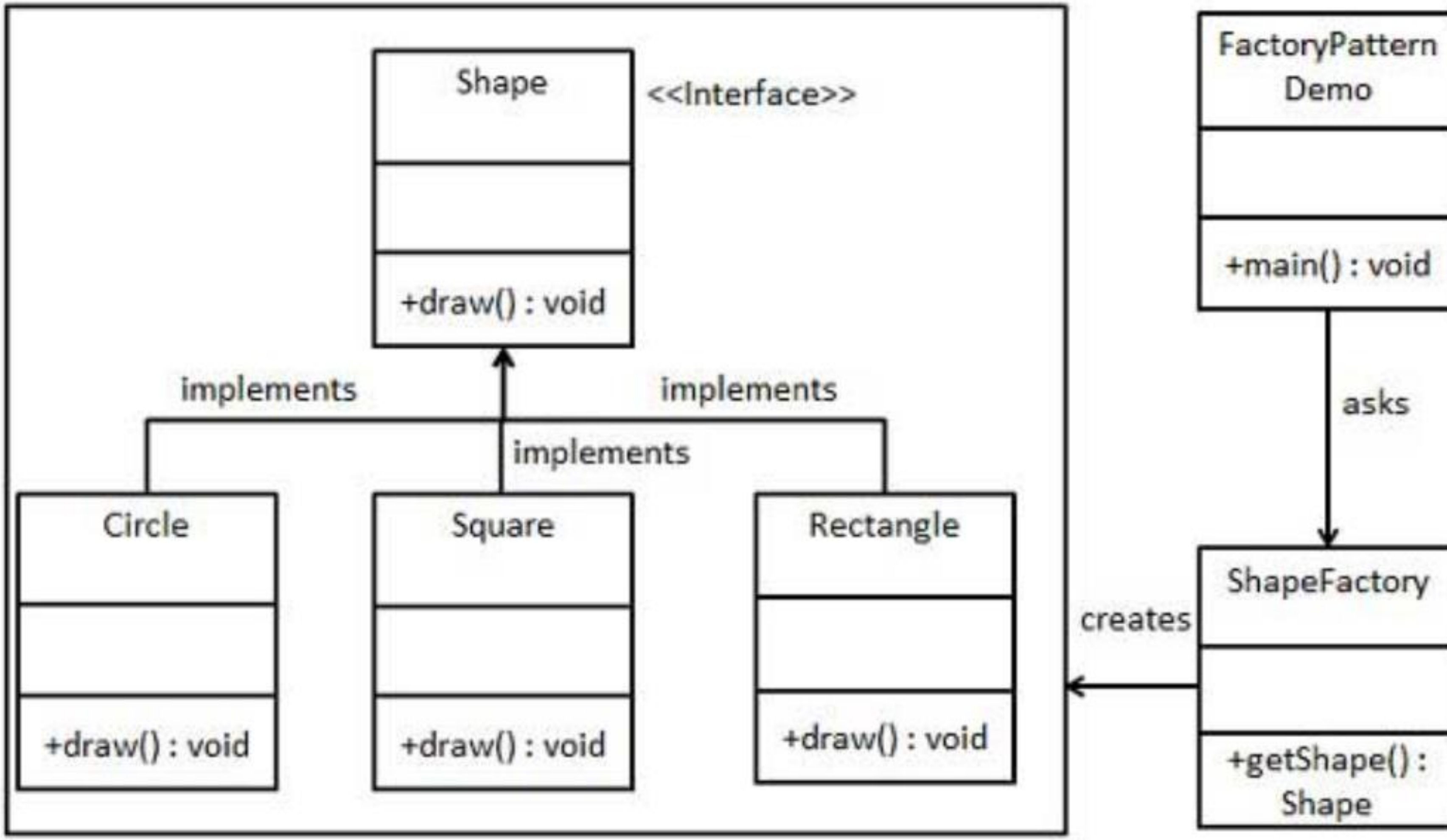
- Factory:

```
class MatrixFactory {
        // not a constructor
        public static Matrix createMatrix() {
                return new SparseMatrix();
        }
}
```

Factory Method

- create objects without exposing the creation logic to the client
- Refer to newly created object using a common interface.

Example: A Shape Factory



https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

```java
public interface Shape {
  void draw();
}

public class Rectangle implements Shape {

  @Override
  public void draw() {
    System.out.println("Inside Rectangle::draw() method.");
  }
}

public class Square implements Shape {

  @Override
  public void draw() {
    System.out.println("Inside Square::draw() method.");
  }
}

// similar for circle etc.
```

```java
public class ShapeFactory {

  //use getShape method to get object of type shape
  public Shape getShape(String shapeType){
    if(shapeType == null){
      return null; // maybe better to throw an exception?
    }
    if(shapeType.equalsIgnoreCase("CIRCLE")){
      return new Circle();

    } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
      return new Rectangle();

    } else if(shapeType.equalsIgnoreCase("SQUARE")){
      return new Square();
    }

    return null;  // maybe better to throw an exception
  }
}
```

```java
public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Rectangle
      shape2.draw();

      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of circle
      shape3.draw();
   }
}
```

An Example

```
class Race {
   Race createRace() {
      Bicycle bike1 = new Bicycle();
      Bicycle bike2 = new Bicycle(); …
   }
}
class TourDeFrance extends Race {
   Race createRace() {
      Bicycle bike1 = new RoadBicycle();
      Bicycle bike2 = new RoadBicycle(); …
   }
}
class Cyclocross extends Race {
   Race createRace() {
      Bicycle bike1 = new MountainBicycle();
      Bicycle bike2 = new MountainBicycle(); …
   }
}
```

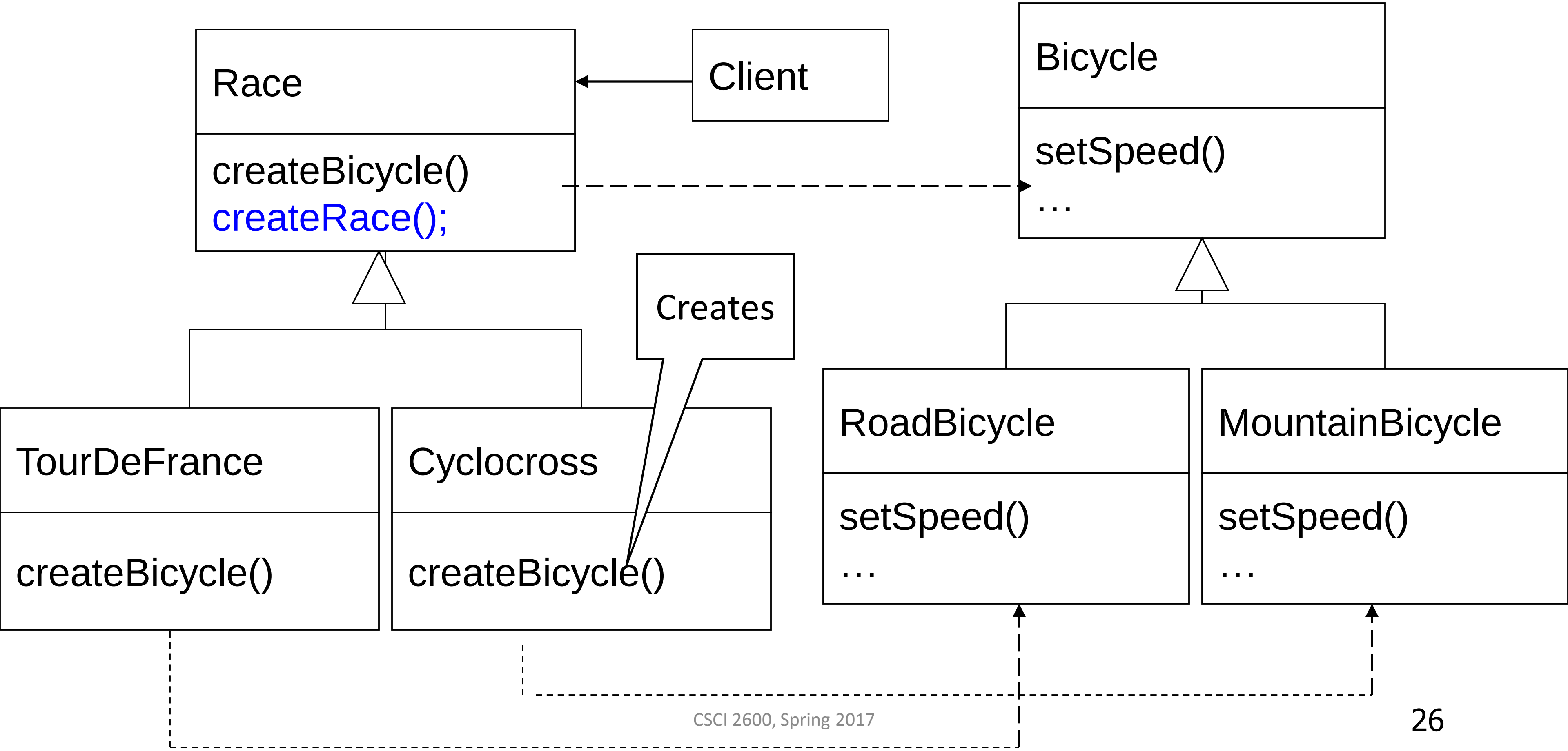
Using a Factory Method

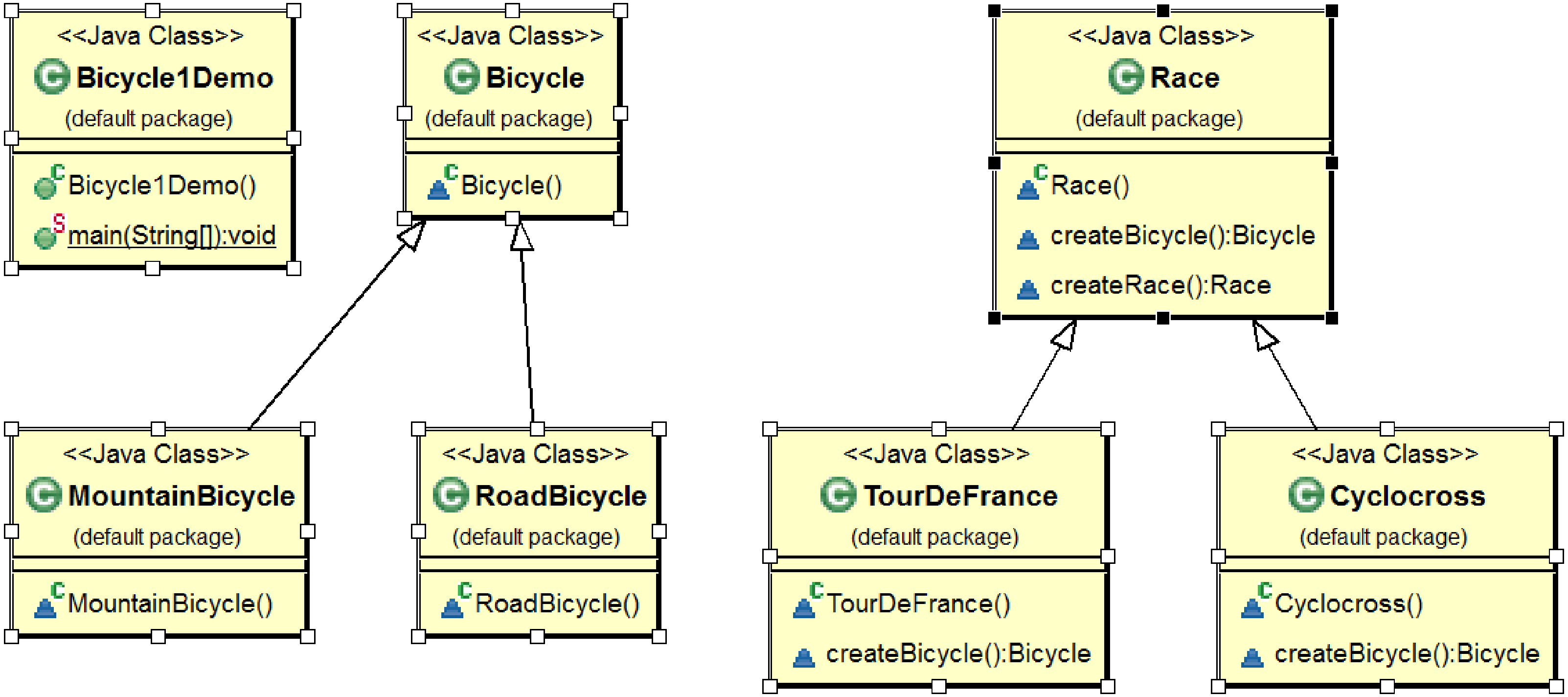
```
class Race {
   Bicycle createBicycle() { … }
   Race createRace() {
      Bicycle bike1 = this.createBicycle();
      Bicycle bike2 = this.createBicycle(); …
   }
}
class TourDeFrance extends Race {
   Bicycle createBicycle() {
      return new RoadBicycle();
   }
}
class Cyclocross extends Race {
   Bicycle createBicycle() {
      return new MountainBicycle();
   }
}
```

Factory

Parallel Hierarchies

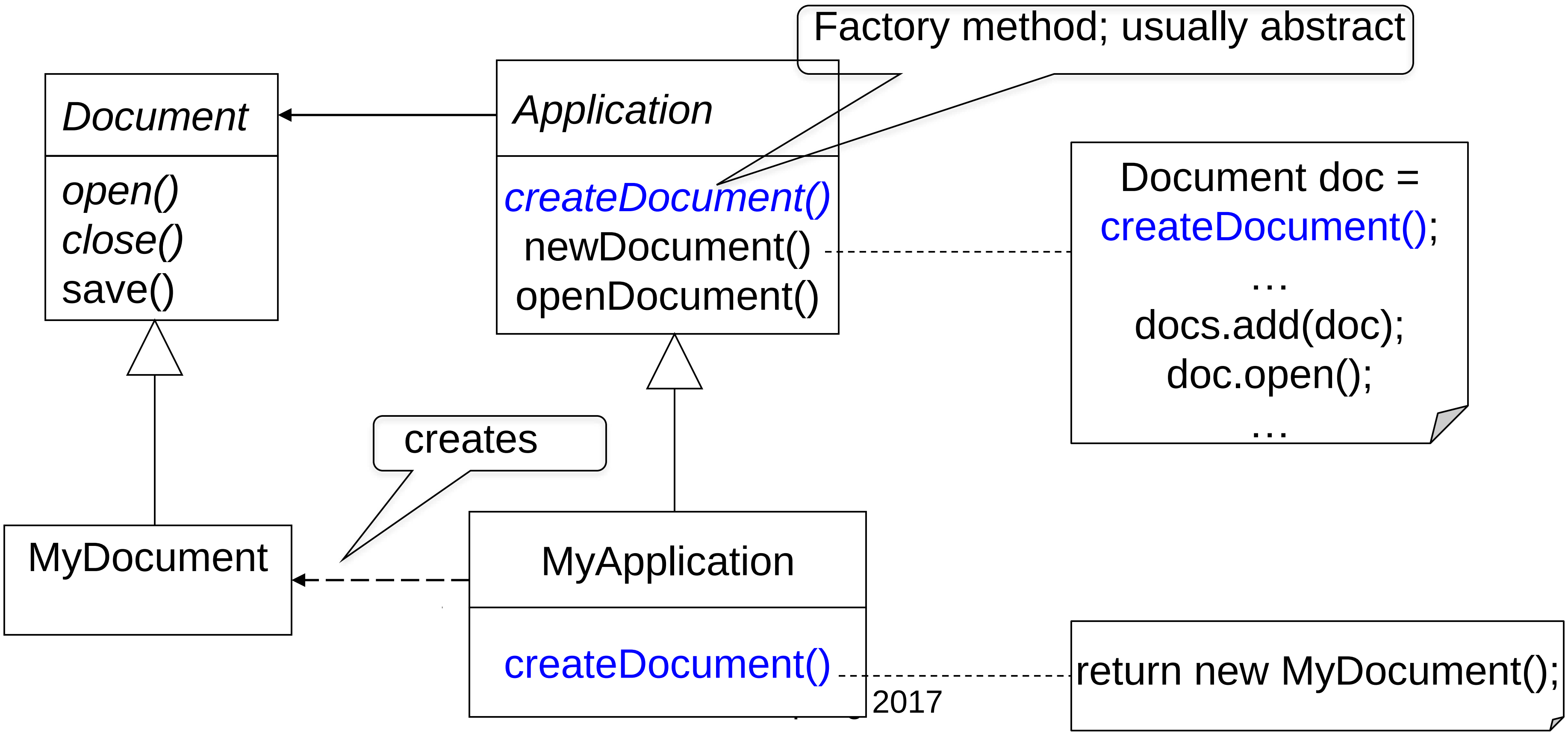
- Can extend with new Races and Bikes with no modification to createRace!




<<Java Class>>
Bicycle1Demo
(default package)
Bicycle1Demo()
main(String[]):void
<<Java Class>>
Bicycle
(default package)
Bicycle()
<<Java Class>>
Race
(default package)
Race()
createBicycle():Bicycle
createRace():Race
<<Java Class>>
MountainBicycle
(default package)
MountainBicycle()
<<Java Class>>
RoadBicycle
(default package)
RoadBicycle()
<<Java Class>>
TourDeFrance
(default package)
TourDeFrance()
createBicycle():Bicycle
<<Java Class>>
Cyclocross
(default package)
Cyclocross()
createBicycle():Bicycle

Another Factory Method Example

- Motivation: Applications share common functions, but work with different documents



Yet Another Factory Method Example

```
abstract class MazeGame {
   abstract Room createRoom();
   abstract Wall createWall();
   abstract Door createDoor();
   Maze createMaze() {
      …
      Room r1 = createRoom(); Room r2 = …
      Wall w1 = createWall(r1,r2);
      Door d1 = createDoor(w1);
      …
   }
   …
}
```

Factory methods

Yet Another Factory Method Example

```
class EnchantedMazeGame extends MazeGame {
    Room createRoom() {
      return new EnchantedRoom(castSpell());
    }
    Wall createWall(Room r1, Room r2) {
      return
       new EnchantedWall(r1,r2,castSpell());
    }
    Door createDoor(Wall w) {
      return new EnchantedDoor(w,castSpell());
    }
}

// Inherits createMaze from MazeGame
```

Yet Another Factory Method Example

```
class BombedMazeGame extends MazeGame {
   Room createRoom() {
     return new RoomWithBomb();
   }
   Wall createWall(Room r1, Room r2) {
     return new BombedWall(r1,r2);
   }
   Door createDoor(Wall w) {
     return new DoorWithBomb(w);
   }
}

// Again, inherit createMaze from MazeGame
```

Factory Methods in the JDK

- **`DateFormat`** class encapsulates knowledge on how to format a **`Date`**
  - Options: Just date? Just time? date+time? where in the world?

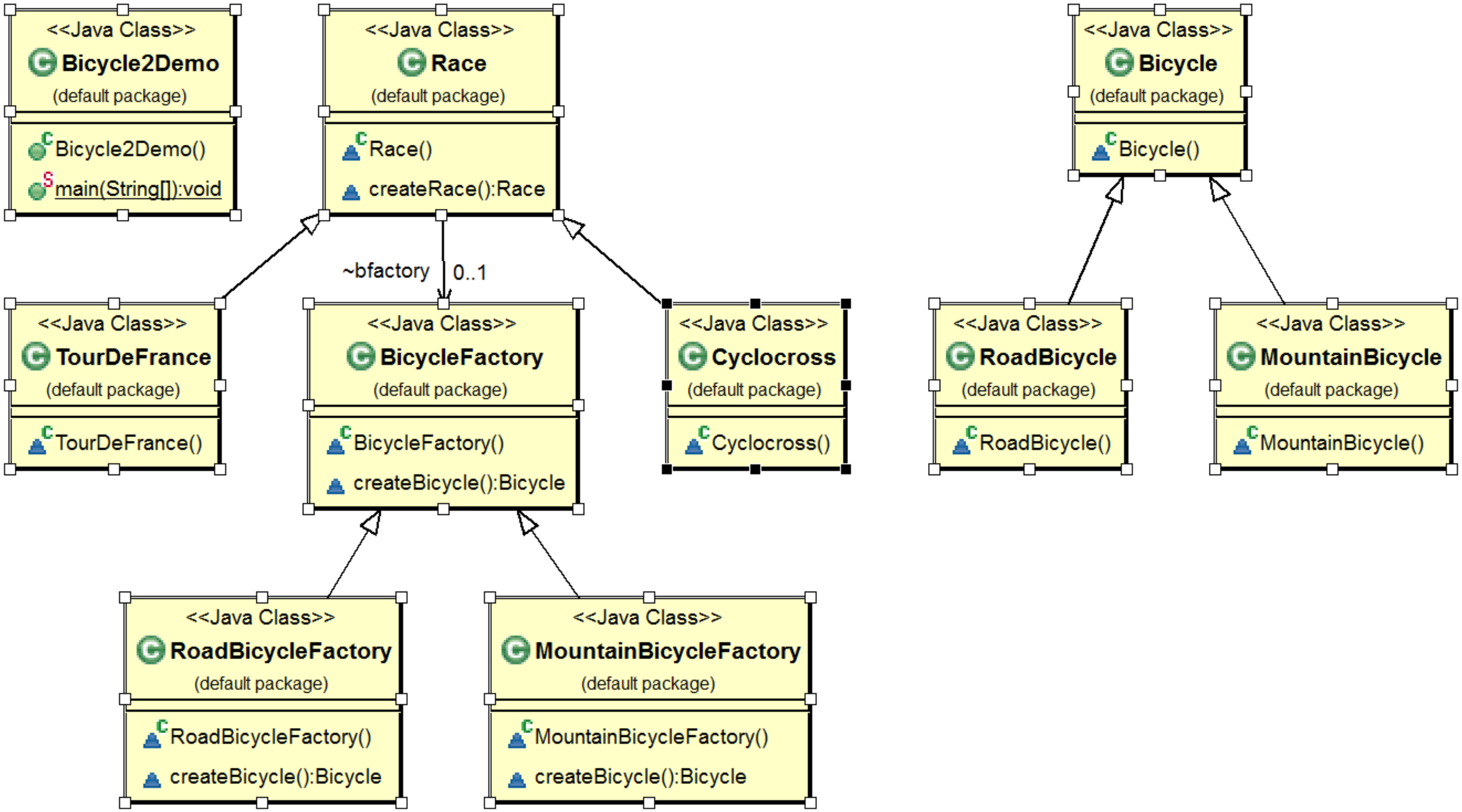
```
DateFormat df1 = DateFormat.getDateInstance();
DateFormat df2 = DateFormat.getTimeInstance();
DateFormat df3 = DateFormat.getDateInstance
  (DateFormat.FULL.Locale.FRANCE);
Date today = new Date();
df1.format(today); // "Jul 4, 1776"
df2.format(today); // "10:15:00 AM"
df3.format(today);  // "jeudi 4 juillet 1776"
```

Bicycle Factory Object

```
class BicycleFactory {
        Bicycle createBicycle() { ... }
        Frame createFrame() { ... }
        Wheel createWheel() { ... }
        ...
        }
class RoadBicycleFactory extends  BicycleFactory {
        Bicycle createBicycle() {
                return new RoadBicycle();
        }
}
class MountainBicycleFactory extends BicycleFactory {
        Bicycle createBicycle() {
                return new MountainBicycle();
        }
}
```

Using a Factory Object

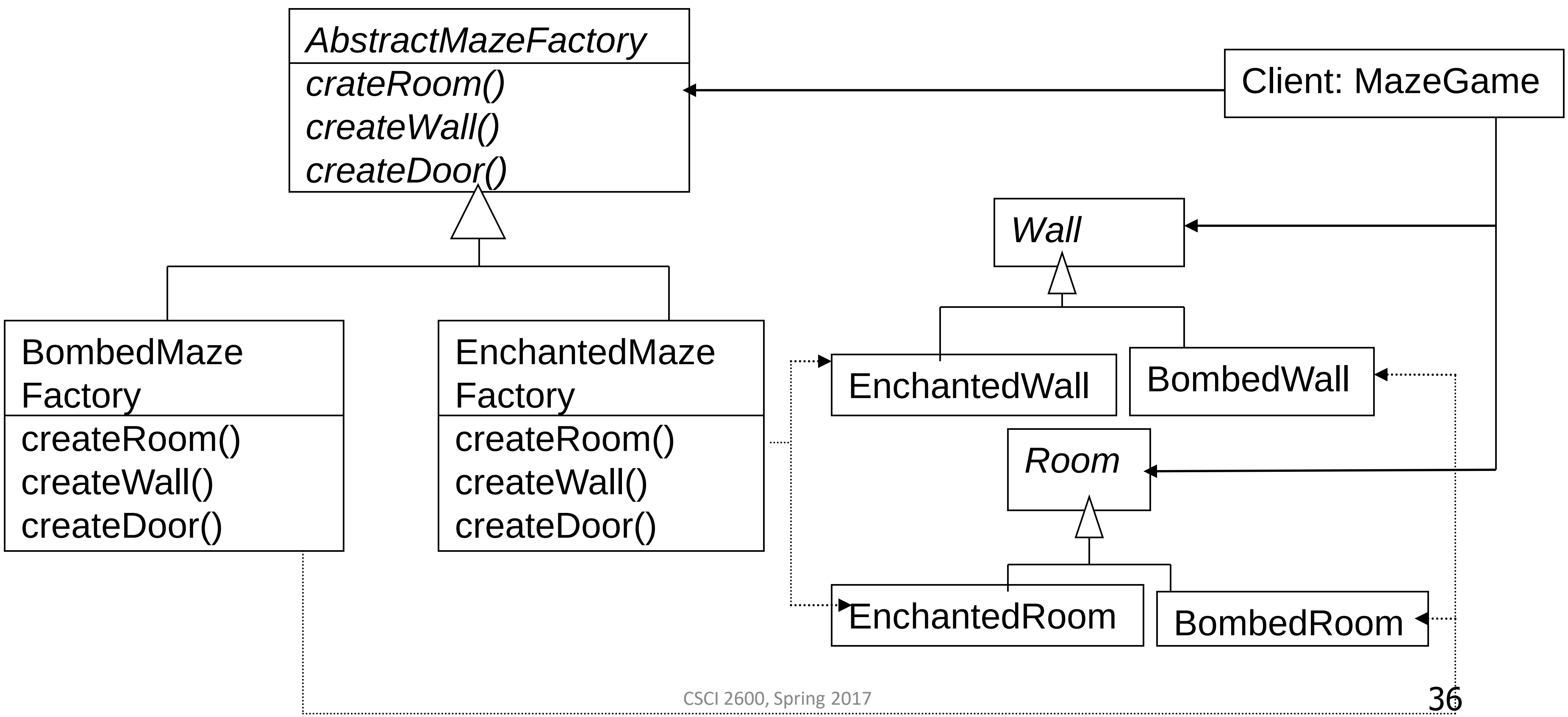
```
class Race {
        BicycleFactory bfactory;
        // constructor
        Race() {
                bfactory = new BicycleFactory();
        }
        Race createRace() {
                Bicycle bike1 = bfactory.createBicycle();
                Bicycle bike2 = bfactory.createBicycle();
                ...
        }
}
class TourDeFrance extends Race {
        // constructor
        TourDeFrance() {
                bfactory = new RoadBicycleFactory();
                ...
        }
}
class Cyclocross extends Race {
        // constructor
        Cyclocross() { bfactory = new MountainBicycleFactory();
        ...
}
```



<<Java Class>>
Bicycle2Demo
(default package)
Bicycle2Demo()
main(String[]):void
<<Java Class>>
Race
(default package)
Race()
createRace():Race
~bfactory
0..1
<<Java Class>>
TourDeFrance
(default package)
TourDeFrance()
<<Java Class>>
BicycleFactory
(default package)
BicycleFactory()
createBicycle():Bicycle
<<Java Class>>
Cyclocross
(default package)
Cyclocross()
<<Java Class>>
RoadBicycleFactory
(default package)
RoadBicycleFactory()
createBicycle():Bicycle
<<Java Class>>
MountainBicycleFactory
(default package)
MountainBicycleFactory()
createBicycle():Bicycle
<<Java Class>>
Bicycle
(default package)
Bicycle()
<<Java Class>>
RoadBicycle
(default package)
RoadBicycle()
<<Java Class>>
MountainBicycle
(default package)
MountainBicycle()

Factory Object Pattern (also known as Abstract Factory)

- Motivation: Encapsulate the factory methods into one class. Separate control over creation



Let's Use a Factory Object

```
class MazeGame {
   AbstractMazeFactory mfactory;
   MazeGame(AbstractMazeFactory mfactory) {
     this.mfactory = mfactory;
   }
   Maze createMaze() {
     …
     Room r1 = mfactory.createRoom();
     Room r2 = …
     Wall w1 = mfactory.createWall(r1,r2);
     Door d1 = mfactory.createDoor(w1);
     …
   }
   …
}
```

The Factory Hierarchy

```
abstract class AbstractMazeFactory {
   Room createRoom();
   Wall createWall(Room r1, Room r2);
   Door createDoor(Wall w);
}
class EnchantedMazeFactory extends
                        AbstractMazeFactory {
   Room createRoom() { creates enchanted … }
   Wall createWall(…) { creates enchanted … }
   Door createDoor(…) { creates enchanted … }
}

class BombedMazeFactory extends
                         AbstractMazeFactory {
   // analogous
}
```

Let's Use Factory Object

```
class Race {
   BikeFactory bfactory;
   Race() { bfactory = new BikeFactory(); }
   Race createRace() {
     Bicycle bike1 = bfactory.createBicycle();
     Bicycle bike2 = bfactory.createBicycle();
     …
   }
}
class TourDeFrance extends Race {
   // constructor
   TourDeFrance() {
     bfactory = new RoadBikeFactory()
   }
}
// analogous constructor for Cyclocross
```

The Factory Hierarchy

Factory methods encapsulated into Factory classes

```
class BikeFactory {
   Bicycle createBicycle() { … }
   Frame createFrame() { … }
   Wheel createWheel() { … }
}
class RoadBikeFactory extends BikeFactory {
   Bicycle crateBicycle() {
      return new RoadBicycle();
   }
}
class MountainBikeFactory extends BikeFactory {
   Bicycle createBicycle() {
      return new MountainBicycle();
   }
}
```

Separate Control Over Races and Bicycles

Control over Bike creation is now passed to BikeFactory

```
class Race {
   BikeFactory bfactory;
   Race(BikeFactory bfactory) {
     this.bfactory = bfactory;
   }
   Race createRace() {
     Bicycle bike1 = bfactory.createBicycle();
     Bicycle bike2 = bfactory.createBicycle();
     …
   }
}
```

- No special constructor for **TourDeFrance** and **Cyclocross**

Separate Control Over Races and Bicycles

- Client can specify the race and the bicycle separately:

```
Race race=new TourDeFrance(new TricycleFactory());
```

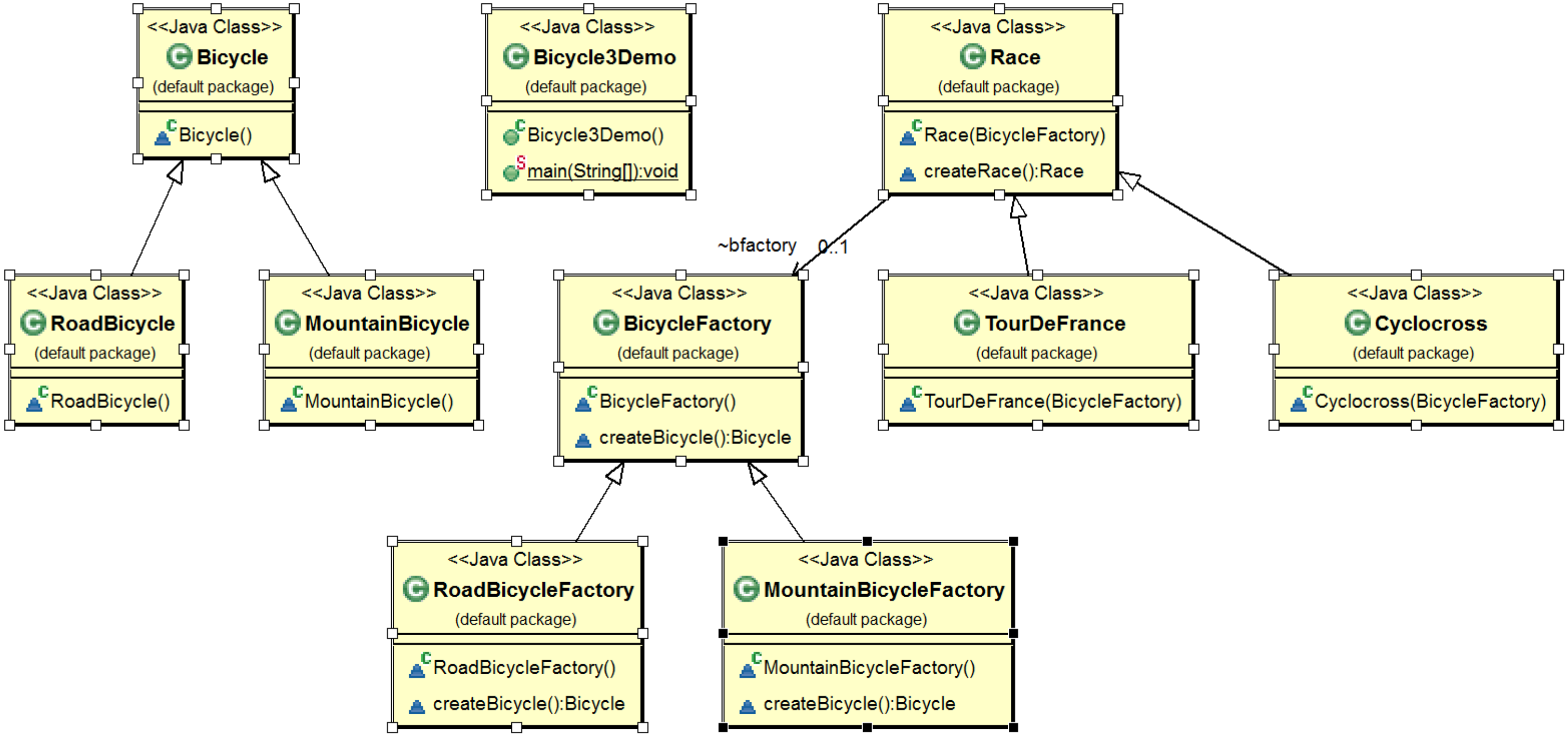
- To specify a different race/bicycle need only change one line:

```
Race race=new Cyclocross(new TricycleFactory());
```

or

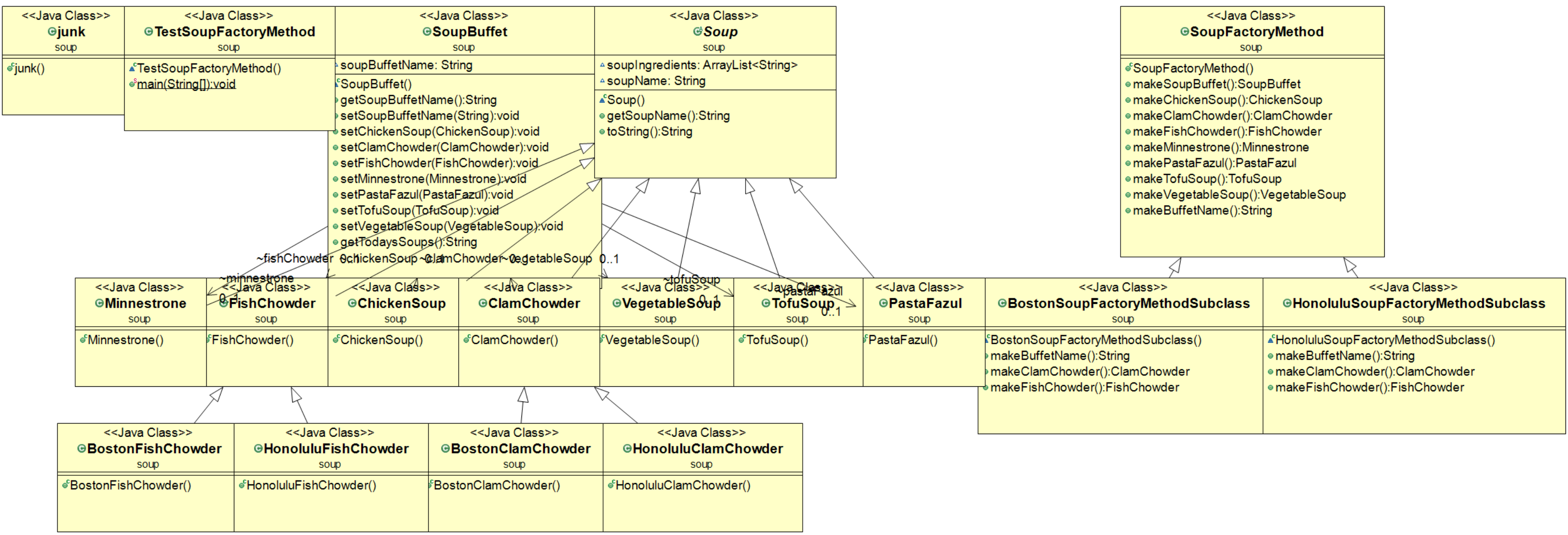
```
Race race=new Cyclocross(new MountainBikeFactory());
```

- Rest of code, uses **Race**, stays the same!


<<Java Class>>
Bicycle
(default package)
Bicycle()
<<Java Class>>
Bicycle3Demo
(default package)
Bicycle3Demo()
main(String[]):void
<<Java Class>>
Race
(default package)
Race(BicycleFactory)
createRace():Race
~bfactory
0..1
<<Java Class>>
RoadBicycle
(default package)
RoadBicycle()
<<Java Class>>
MountainBicycle
(default package)
MountainBicycle()
<<Java Class>>
BicycleFactory
(default package)
BicycleFactory()
createBicycle():Bicycle
<<Java Class>>
TourDeFrance
(default package)
TourDeFrance(BicycleFactory)
<<Java Class>>
Cyclocross
(default package)
Cyclocross(BicycleFactory)
<<Java Class>>
RoadBicycleFactory
(default package)
RoadBicycleFactory()
createBicycle():Bicycle
<<Java Class>>
MountainBicycleFactory
(default package)
MountainBicycleFactory()
createBicycle():Bicycle

Extended Example

- http://www.fluffycat.com/Java-Design-Patterns/Factory-Method/
- Extended example of a Soup Factory



<<Java Class>>
junk
soup
junk()
<<Java Class>>
TestSoupFactoryMethod
soup
TestSoupFactoryMethod()
main(String[]):void
<<Java Class>>
SoupBuffet
soup
soupBuffetName: String
SoupBuffet()
getSoupBuffetName():String
setSoupBuffetName(String):void
setChickenSoup(ChickenSoup):void
setClamChowder(ClamChowder):void
setFishChowder(FishChowder):void
setMinnestrone(Minnestrone):void
setPastaFazul(PastaFazul):void
setTofuSoup(TofuSoup):void
setVegetableSoup(VegetableSoup):void
getTodaysSoups():String
<<Java Class>>
Soup
soup
soupIngredients: ArrayList<String>
soupName: String
Soup()
getSoupName():String
toString():String
<<Java Class>>
SoupFactoryMethod
soup
SoupFactoryMethod()
makeSoupBuffet():SoupBuffet
makeChickenSoup():ChickenSoup
makeClamChowder():ClamChowder
makeFishChowder():FishChowder
makeMinnestrone():Minnestrone
makePastaFazul():PastaFazul
makeTofuSoup():TofuSoup
makeVegetableSoup():VegetableSoup
makeBuffetName():String
~fishChowder 0..1
~chickenSoup 0..1
~clamChowder 0..1
~vegetableSoup 0..1
~minnestrone 0..1
~tofuSoup 0..1
~pastaFazul 0..1
<<Java Class>>
Minnestrone
soup
Minnestrone()
<<Java Class>>
FishChowder
soup
FishChowder()
<<Java Class>>
ChickenSoup
soup
ChickenSoup()
<<Java Class>>
ClamChowder
soup
ClamChowder()
<<Java Class>>
VegetableSoup
soup
VegetableSoup()
<<Java Class>>
TofuSoup
soup
TofuSoup()
<<Java Class>>
PastaFazul
soup
PastaFazul()
<<Java Class>>
BostonSoupFactoryMethodSubclass
soup
BostonSoupFactoryMethodSubclass()
makeBuffetName():String
makeClamChowder():ClamChowder
makeFishChowder():FishChowder
<<Java Class>>
HonoluluSoupFactoryMethodSubclass
soup
HonoluluSoupFactoryMethodSubclass()
makeBuffetName():String
makeClamChowder():ClamChowder
makeFishChowder():FishChowder
<<Java Class>>
BostonFishChowder
soup
BostonFishChowder()
<<Java Class>>
HonoluluFishChowder
soup
HonoluluFishChowder()
<<Java Class>>
BostonClamChowder
soup
BostonClamChowder()
<<Java Class>>
HonoluluClamChowder
soup
HonoluluClamChowder()

Dependency Injection

- In Java, we can decide what **`Factory`** to initialize with at runtime!
- External dependency injection:

```
BikeFactory f = (BikeFactory)
         DepenencyManager.get("BikeFactory");
Race race = new Cyclocross(f);
```

- An external file specifies a value for "BikeFactory", factory in plain text, say "TricycleFactory"
- **`DependencyManager`** reads file and uses Java reflection to load and instantiate class, **`TricycleFactory`**

Dependency Injection

- String factory = // String read from file
- Class factoryClazz = Class.forName(factory);
- BikeFactory bfactory = (BikeFactory) factoryClazz.newInstance();

Factory Pattern

- Factory pattern encapsulates creation of different variations of objects
  - Factory method
  - Factory object
- Helps overcome limitations of object constructors

The Prototype Pattern

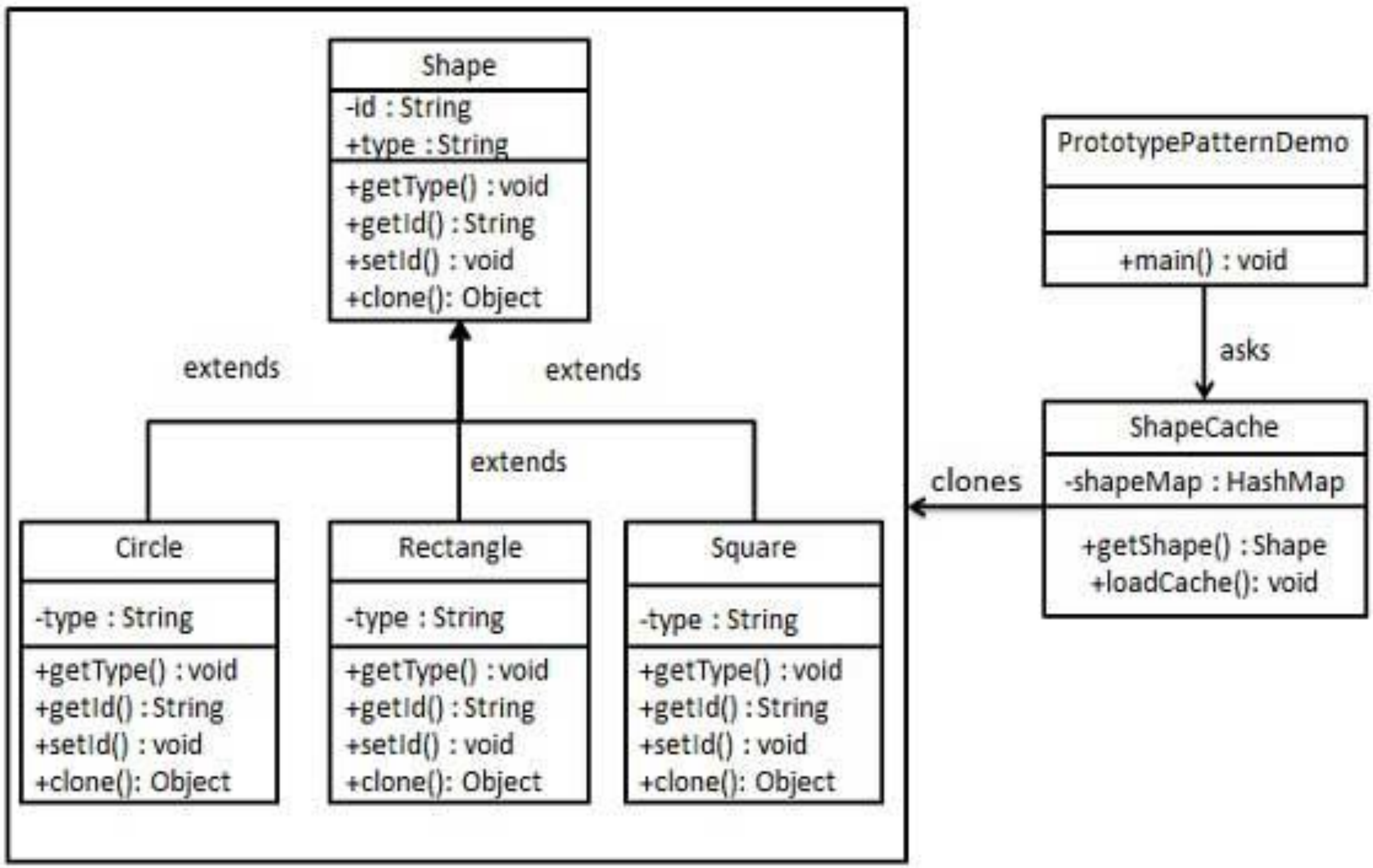
- Every object itself is a factory
- Each class can define a **`clone`** method that returns a copy of the receiver object

```
class Bicycle {
   Bicycle clone() { … }
}
```

- Often **`Object`** is the return type of **`clone`**
  - **`Object`** declares **`protected Object clone()`**
  - In Java 1.4 and earlier an overriding method cannot change the return type. Now an overriding method can change it covariantly

Prototype

- Prototype interface creates a clone of the current object.
- Useful when creation of the object directly is costly.
  - For example, an object is to be created after a costly database operation.
  - Cache the object
  - returns its clone on next request
  - only update the database when needed
  - Reduces database calls.



https://www.tutorialspoint.com/design_pattern/prototype_pattern.htm

```java
public abstract class Shape implements Cloneable {

  private String id;
  protected String type;

  abstract void draw();

  public String getType(){
    return type;
  }

  public String getId() {
    return id;
  }

  public void setId(String id) {
    this.id = id;
  }

  public Object clone() {
    Object clone = null;

    try {
      clone = super.clone();
    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
    }

    return clone;
  }
}
```

```java
public class Rectangle extends Shape {

   public Rectangle(){
     type = "Rectangle";
   }

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}

// similar classes for Circle and Square
```

```java
import java.util.Hashtable;

public class ShapeCache {

  private static Hashtable<String, Shape> shapeMap  = new Hashtable<String, Shape>();

  public static Shape getShape(String shapeId) {
    Shape cachedShape = shapeMap.get(shapeId);
    return (Shape) cachedShape.clone();
  }

  // for each shape run database query and create shape
  // shapeMap.put(shapeKey, shape);
  // for example, we are adding three shapes

  public static void loadCache() {
    Circle circle = new Circle();
    circle.setId("1");
    shapeMap.put(circle.getId(),circle);

    Square square = new Square();
    square.setId("2");
    shapeMap.put(square.getId(),square);

    Rectangle rectangle = new Rectangle();
    rectangle.setId("3");
    shapeMap.put(rectangle.getId(), rectangle);
  }
}
```

```
public class PrototypePatternDemo {
  public static void main(String[] args) {
    ShapeCache.loadCache();

    Shape clonedShape = (Shape) ShapeCache.getShape("1");
    System.out.println("Shape : " + clonedShape.getType());

    Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
    System.out.println("Shape : " + clonedShape2.getType());

    Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
    System.out.println("Shape : " + clonedShape3.getType());
  }
}
```

Output:
Shape : Circle
Shape : Square
Shape : Rectangle

Using Prototypes

```
class Race {
   Bicycle bproto;
   // constructor
   Race(Bicycle bproto) {
     this.bproto = bproto;
   }
   Race createRace() {
     Bicycle bike1 = bproto.clone();
     Bicycle bike2 = bproto.clone();
     …
   }
}
```

How do we specify the race and the bicycle?

```
new TourDeFrance(new Tricycle());
```

Sharing

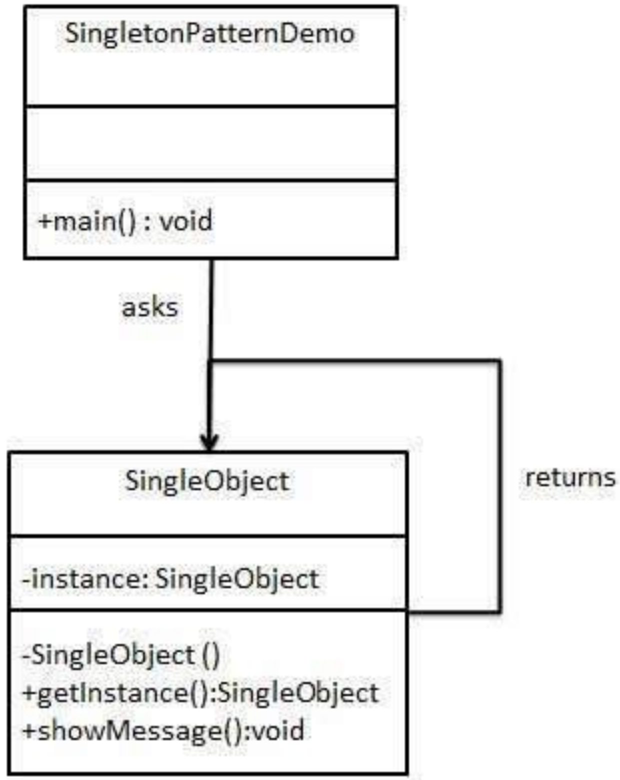
- Recall that constructors always return a new object, never a pre-existing one
- In many situations, we would like a pre-existing object
- Singleton pattern: only one object ever exists
  - A factory object is almost always a singleton
- Interning pattern: only one object with a given abstract value exist

Singleton Pattern

- Motivation: there must be a single instance of the class

Factory method --- it produces the instance of the class

```
class Bank {
   private Bank() { … }
   private static Bank instance;
   public static Bank getInstance() {
       if (instance == null)
           instance = new Bank();
       return instance;
   }
   // methods of Bank
}
```



https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm

```java
public class SingleObject {

  //create an object of SingleObject
  // executed when object is loaded
  private static SingleObject instance = new SingleObject();

  //make the constructor private so that this class cannot be
  //instantiated
  private SingleObject(){}

  //Get the only object available
  public static SingleObject getInstance(){
    return instance;
  }

  public void showMessage(){
    System.out.println("Hello World!");
  }
}
```

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

    //illegal construct
    //Compile Time Error: The constructor SingleObject() is not visible
    //SingleObject object = new SingleObject();

    //Get the only object available
    SingleObject object = SingleObject.getInstance();

    //show the message
    object.showMessage();
  }
}
```

Another Singleton Example

Static initializer --- executed when class is loaded

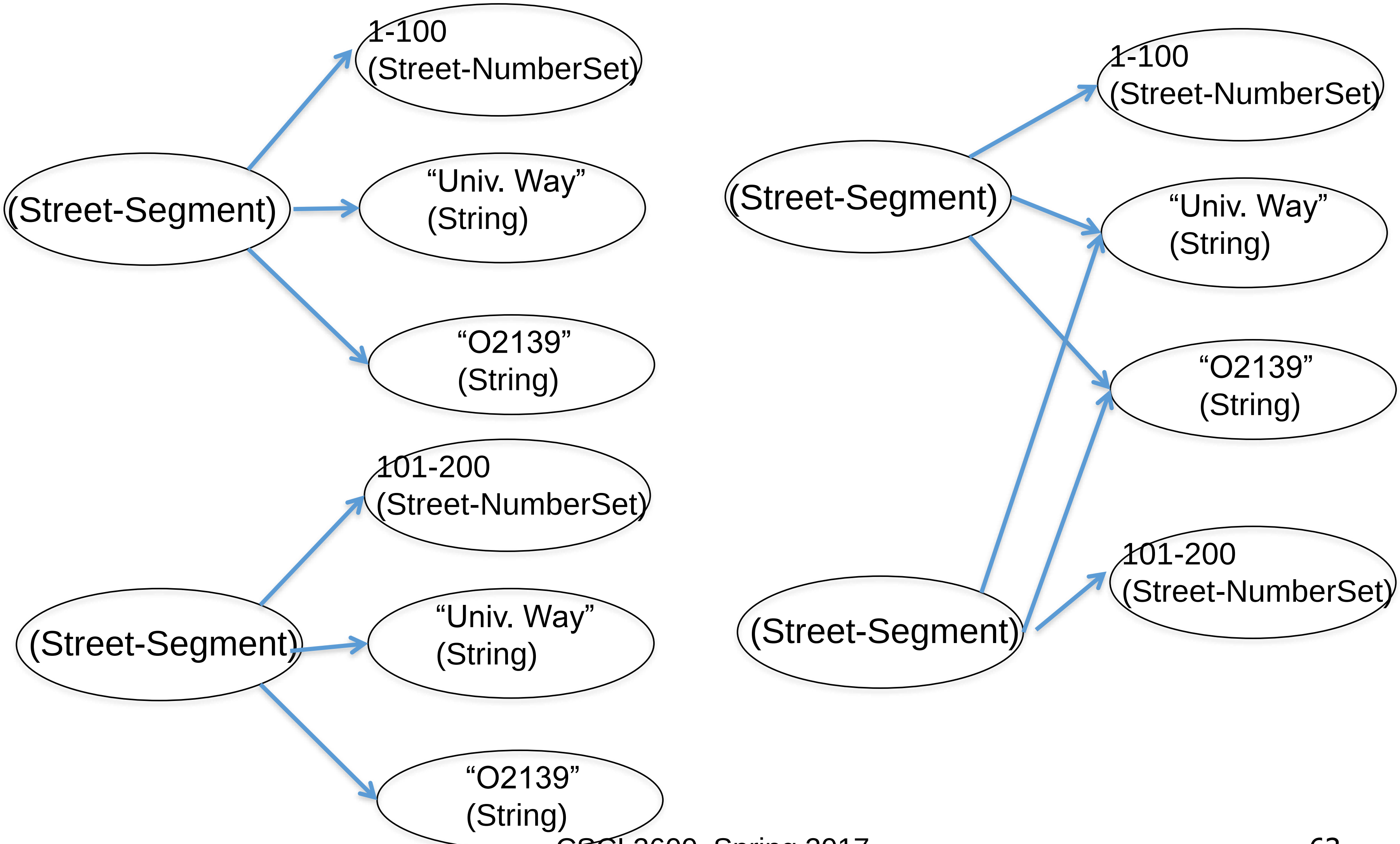
```
public class UserDatabaseSource
            implements UserDatbase {
 private static UserDatabase theInstance =
            new UserDatabaseSource();
  private UserDatabaseSource() { … }
  public static UserDatabase getInstance() {
     return theInstance; }

  public User readUser(String username) { … }
  public void writeUser(User user) { … }
}
```

Interning Pattern

- Not a GoF design pattern
- Reuse existing object with same value, instead of creating a new one
  - E.g., why create multiple Strings “car”? Create a single instance of String “car”!
  - Less space
  - May compare with == instead of **`equals`** and speed the program up
- Interning applied to <u>immutable</u> objects only

Interning Pattern



Interning Pattern

Why not a HashSet but HashMap?

- Maintain a collection of all names
- If an object already exists return that object

```
HashMap<String,String> names;
String canonicalName(String n) {
   if (names.containsKey(n))
     return names.get(n);
   else {
     names.put(n,n);
     return n;
   }
}
```

- Java supports interning for Strings:

`s.intern()` returns a canonical representation of **`s`**

Java Strings Can be Interned

```
public static void main(String[] args) {
        String a = "cat";
        String b = "cat";
        String c = new String("cat");

        System.out.println(a == b);   // prints true
        System.out.println(a.equals(b));  // prints true

        System.out.println(a == c);   // prints false
        System.out.println(a.equals(c));  // prints true
}
```

JavaDoc for String.intern()

public String intern()

Returns a canonical representation for the string object.

A pool of strings, initially empty, is maintained privately by the class String.

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

It follows that for any two strings s and t, s.intern() == t.intern() is true if and only if s.equals(t) is true.

All literal strings and string-valued constant expressions are interned.
String literals are defined in section 3.10.5 of the The Java™ Language Specification.

Returns:
a string that has the same contents as this string, but is guaranteed to be from a pool of unique strings.

Why Not HashSet?

- Maintain a collection of all names
- If an object already exists return that object

```
HashSet<String> names;
String canonicalName(String n) {
   if (names.contains(n))
     return n;
   else {
     names.add(n);
     return n;
   }
}
```

What's wrong with java.lang.Boolean?

```
public class Boolean {
 private final boolean value;
 public Boolean(boolean value) {
   this.value = value;
 }
 public static Boolean FALSE=new Boolean(false);
 public static Boolean TRUE=new Boolean(true);
 public static Boolean valueOf(boolean value) {
   if (value) return TRUE;
   else return FALSE;
 }
}
```

Factory method --- produces the appropriate instance

What's wrong with java.lang.Boolean?

- Boolean constructor should have been private: would have forced interning through **`valueOf`**
- Spec warns against using the constructor
- Joshua Bloch, lead designer of many Java libraries, in 2004: The Boolean type should not have had public constructors. There's really no great advantage to allow multiple trues or multiple falses, and I've seen programs that produce millions of trues and millions of falses creating needless work for the garbage collector.

So, in the case of immutables, I think factory methods are great.

What’s wrong with java.lang.Boolean?

- Note: It is rarely appropriate to use this constructor. Unless a new instance is required,
- The static factory valueOf(boolean) is generally a better choice.
- It is likely to yield significantly better space and time performance.