Design Patterns

Outline

- The Unified Modeling Language (UML)

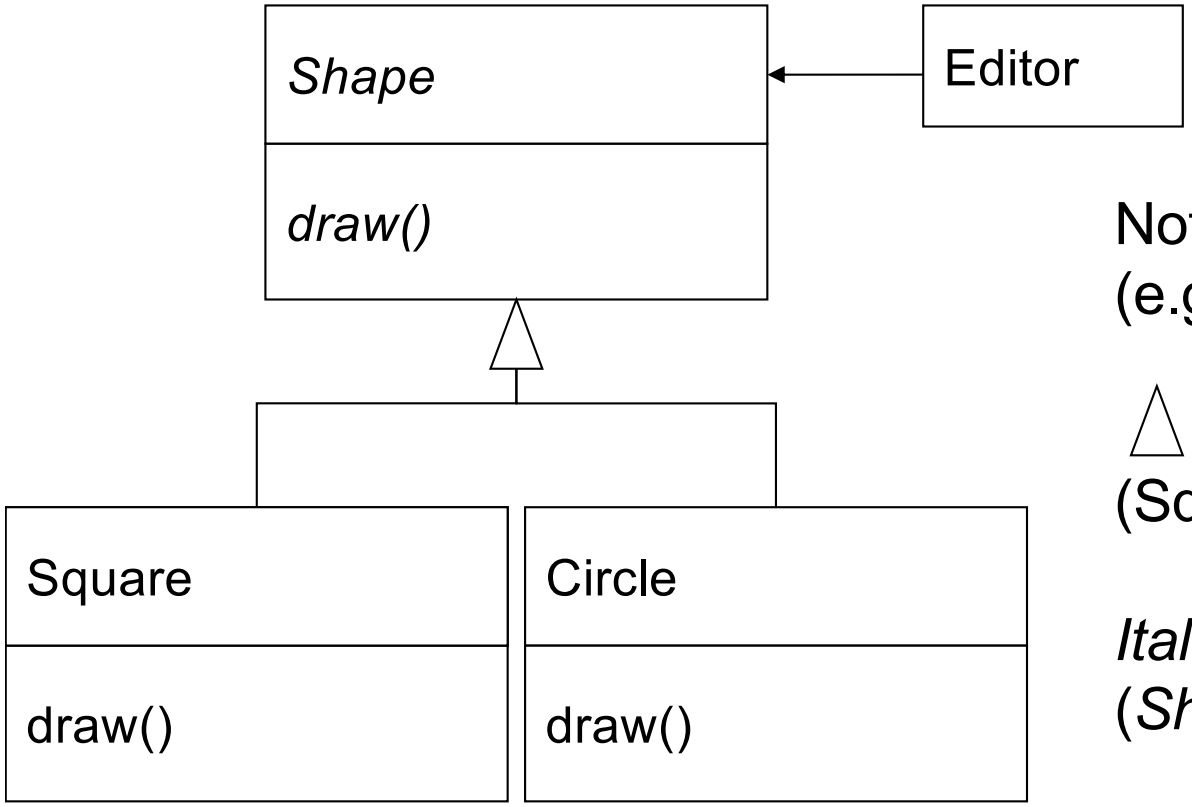
- Design patterns
  - Intro to design patterns
  - Creational patterns
    - Factories: Factory method, Factory object, Prototype
    - Sharing: Singleton, Interning
  - Structural patterns
    - Adapter, Composite, Decorator, Proxy

UML Class Diagrams

- Unified Modeling Language (UML) is the "lingua franca" of object-oriented modeling and design
  - Common language

- UML class diagrams show classes, their interrelationships (inheritance and composition), their attributes and operations
- UML sequence diagrams can also show dynamics of the system

Classes and Inheritance

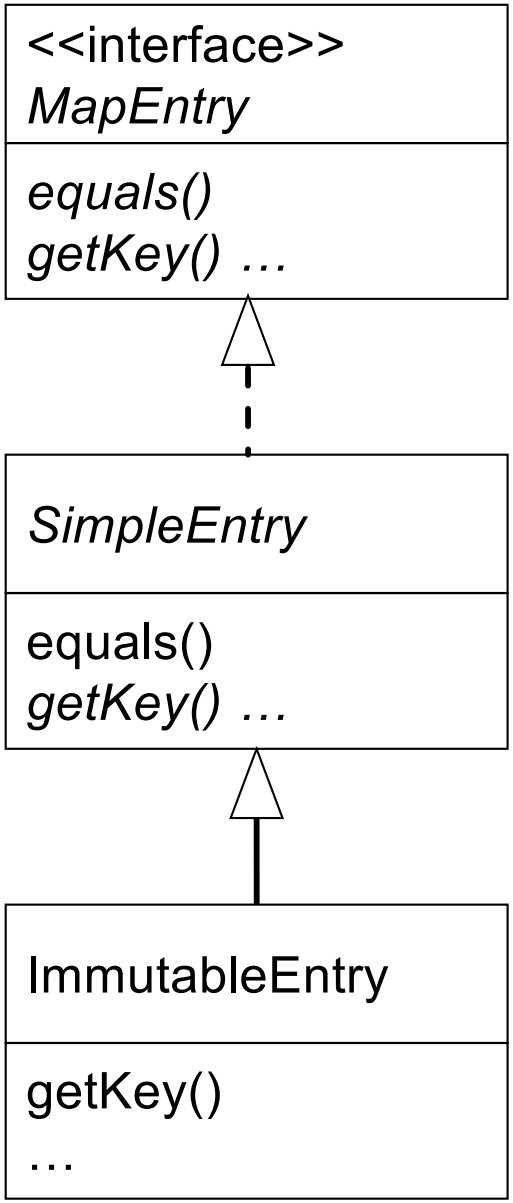


Notation: Boxes are classes (e.g., Shape, Circle).

△ denotes inheritance (Square is a subclass of Shape.)

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

Classes and Inheritance

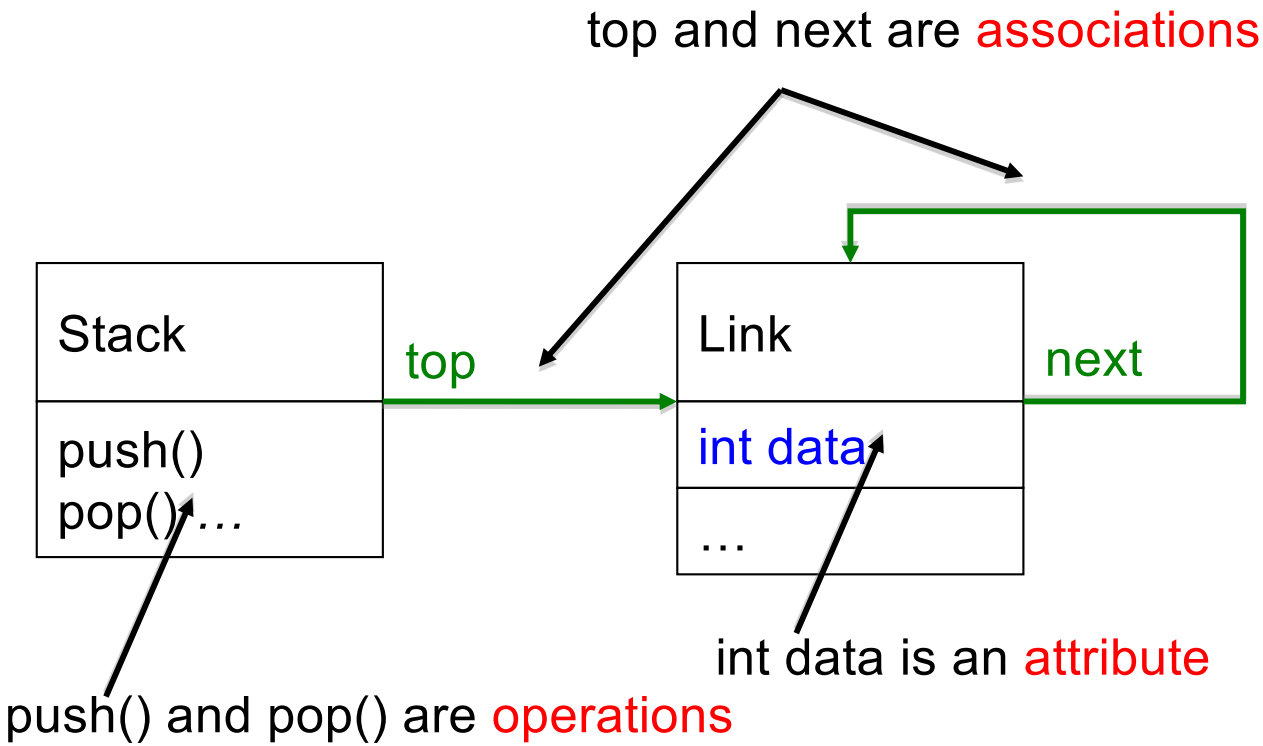


denotes interface inheritance (*SimpleEntry* implements interface *MapEntry*)

ImmutableEntry extends abstract class *SimpleEntry*

Associations

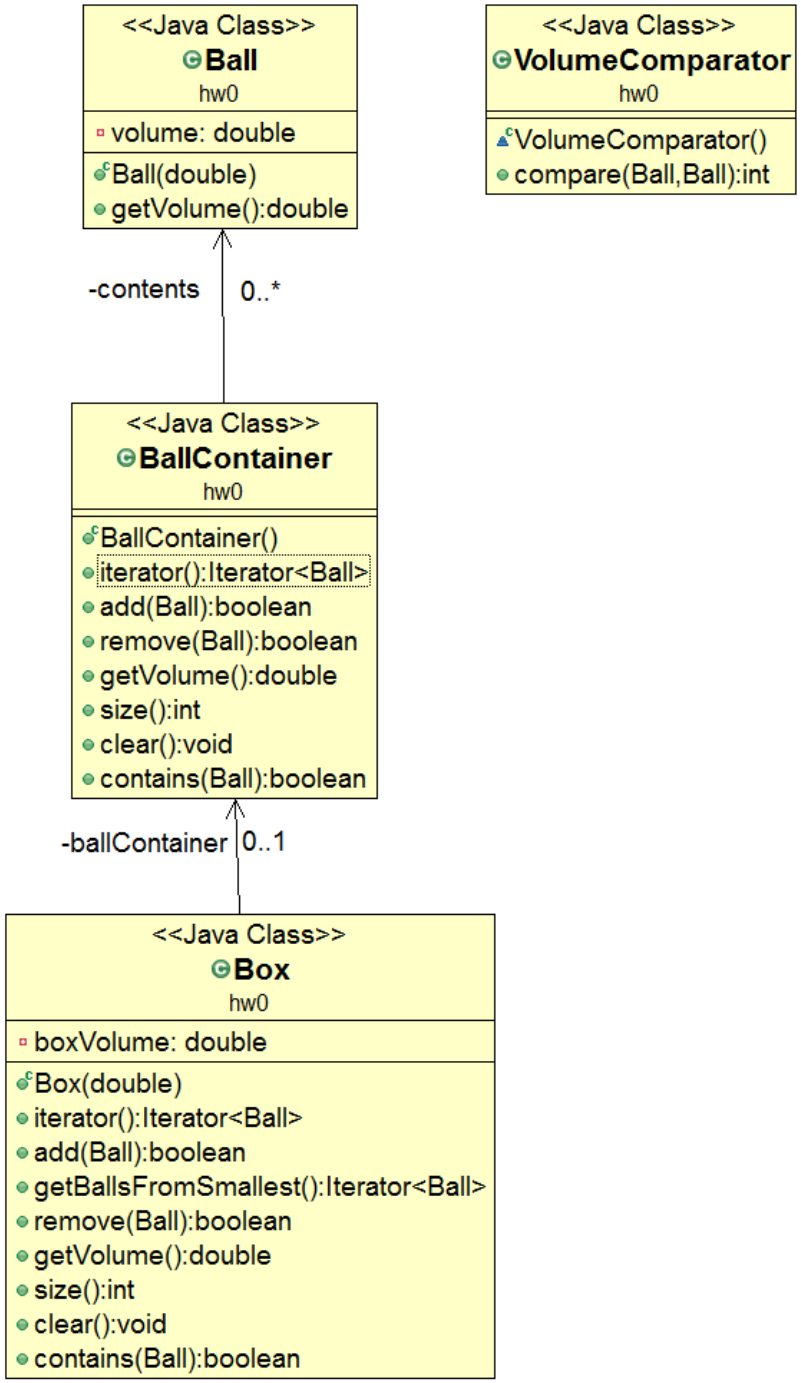
```
class Stack {
    Link top;
    …
}
class Link {
    Link next;
    int data;
    …
}
```

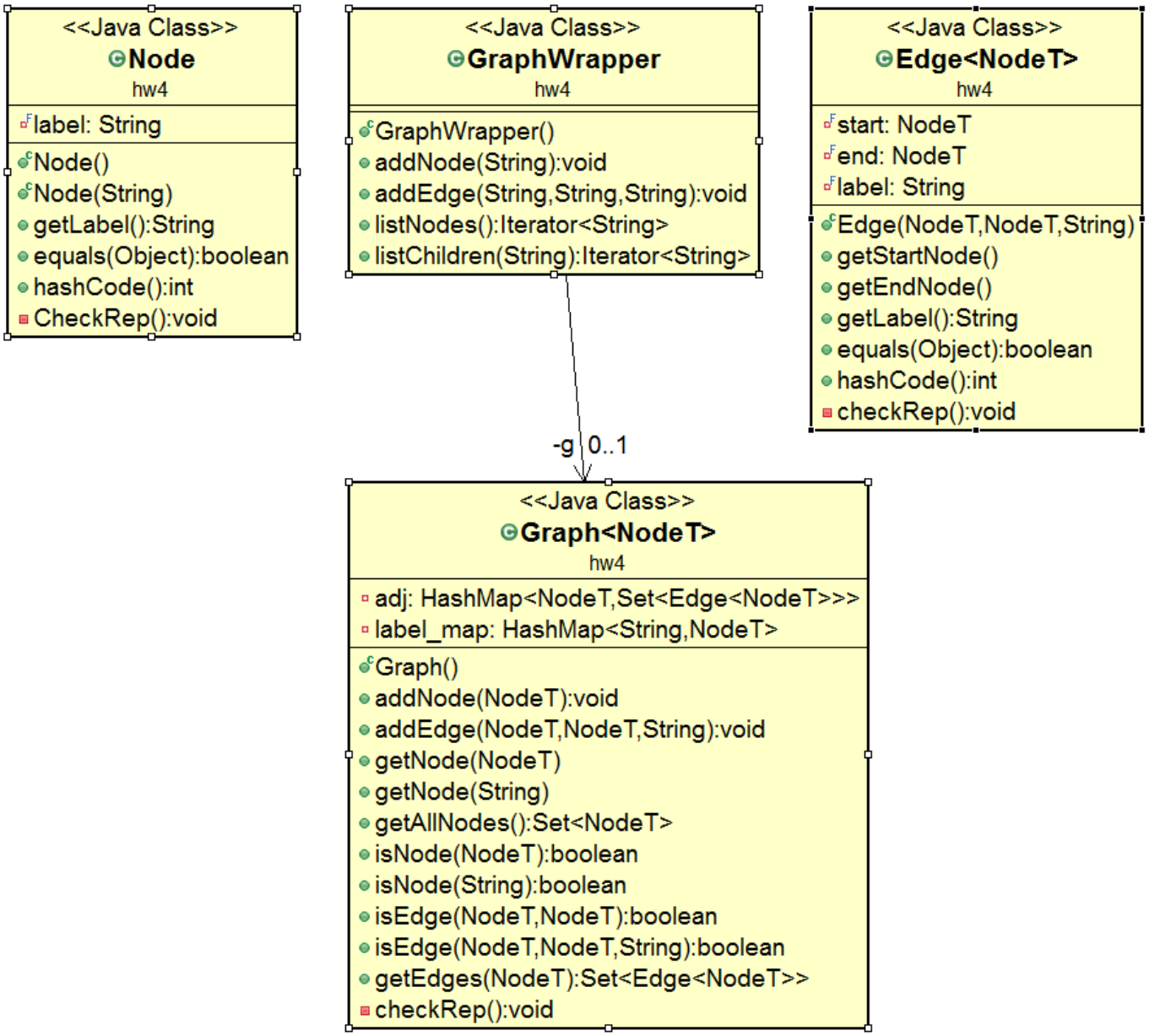


A UML association often represents a composition relationship: object of one class encloses the object(s) of the other. Above, Stack -> Link is a typical composition (has-a) relationship --- the Stack encloses/encapsulates its Link.

Exercise

- Draw a UML class diagram that shows the interrelationships between the classes from HW0
- ObjectAid
  - http://objectaid.com/home
  - Add-on for Eclipse to draw class diagrams
- There are many tools available for drawing, editing, and generating code from UML


<<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

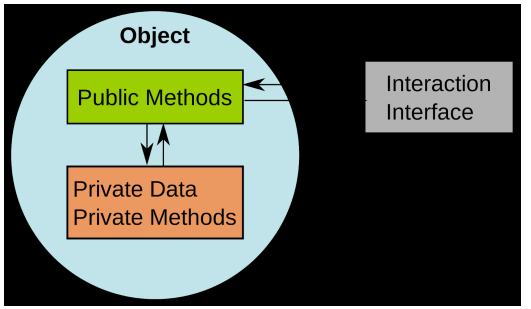
UML

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

Solutions to common programming problems

- Programming solutions
  - Encapsulation
  - Subclassing
  - Iteration
  - Exceptions
  - Generics

Encapsulation (information hiding)

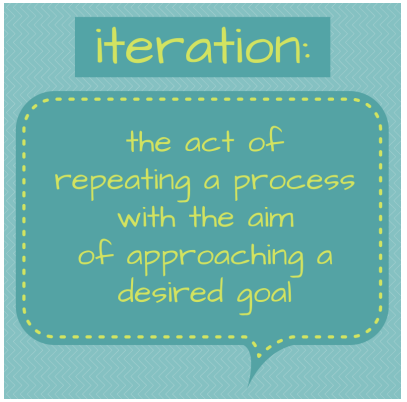


- Problem: Exposed fields can be directly manipulated
  - Violations of the representation invariant
  - Dependences prevent changing the implementation
- Solution:
  - Hide some components
  - Constrain ways to access the object
- Disadvantages:
  - Interface may not (efficiently) provide all desired operations
  - Indirection may reduce performance

Subclassing (inheritance)

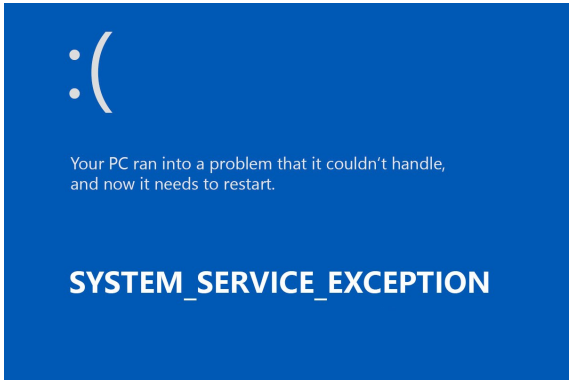
- Problem: Repetition in implementations
  - Similar abstractions have similar members (fields, methods)
- Solution:
  - Inherit default members from a superclass
  - Select an implementation via run-time dispatching
- Disadvantages:
  - Code for a class is spread out, and thus less understandable
  - Run-time dispatching introduces slight overhead

Iteration/Recursion



- Problem: To access all members of a collection, must perform a specialized traversal for each data structure
  - Introduces undesirable dependences
  - Does not generalize to other collections
- Solution:
  - The implementation performs traversals, does bookkeeping
  - The implementation has knowledge about the representation
  - Results are communicated to clients via a standard interface (e.g., hasNext(), next())
- Disadvantages:
  - Iteration order is fixed by the implementation and not under the control of the client

Exceptions



- Problem: Errors in one part of the code should be handled elsewhere.
  - Code should not be cluttered with error-handling code.
  - Return values should not be preempted by error codes.
- Solution:
  - Language structures for throwing and catching exceptions
- Disadvantages
  - Code may still be cluttered.
  - It may be hard to know where an exception will be handled.
  - Use of exceptions for normal control flow may be confusing and inefficient.

Generics

- Problem: Well-designed data structures hold one type of object
- Solution:
  - Programming language checks for errors in contents
  - List<Date> instead of just List
- Disadvantages:
  - More verbose types

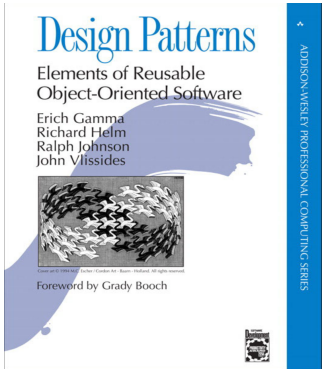
Other problems

- Reuse implementation without subtyping
- Reuse implementation, but change interface
- Permit a class to be instantiated only once
- Constructor that might return an existing object
- Constructor that might return a subclass object

Design Patterns

- A design pattern is a solution to a design problem that occurs over and over again

- The reference: 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
- Design patterns are key in OOP
- SOLID + Design Patterns
  - **S**RP The Single Responsibility Principle: -- a class should have one, and only one, responsibility.
  - **O**CP The Open Closed Principle: -- you should be able to extend a class's behavior, without modifying it.
  - **L**SP The Liskov Substitution Principle: -- derived classes must be substitutable for their base classes.
  - **I**SP The Interface Segregation Principle: -- Many client-specific interfaces are better than one general-purpose interface
  - **D**IP The Dependency Inversion Principle -- depend on abstractions not on concrete implementations.

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 designed software uses design patterns extensively
  - Understanding software requires knowledge of design patterns
- Be careful
  - Design patterns are descriptive, not prescriptive
- Give a common vocabulary for describing programming solutions

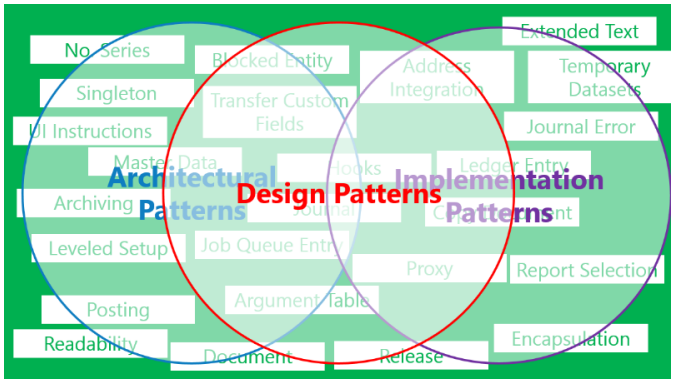
Why Should You Care?

- Programming languages do not have built-in solutions to every problem
- Best solution depends on context
- Every language has shortcomings
  - So does every paradigm: OO, functional, declarative, ...
- Language features often start out as design patterns

Design Patterns Don’t Solve All Problems

- But, they can help
  - Get something basic working first
  - Improve it once you understand it - refactor
- Design patterns can increase or decrease understandability
  - ✓ Improve modularity, separate concerns, ease description
  - ❖ Add indirection, increase code size
  - ➢ Use whatever increases maintainability and efficiency
- 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, 2018.

Design Patterns



- Three categories
  - Creational patterns
    - Constructing objects
    - Abstract Factory, Builder, Factory, Prototype, Singleton
  - Structural patterns
    - Combining objects, control object structure
    - Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy
  - Behavioral patterns
    - Control object behavior
    - Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template, Visitor

Creational Patterns

- Problem: constructors in Java (and other OO languages) are inflexible
    1. Can't create a subtype of the type to which they belong
    2. Always return a fresh new object, can't reuse the same object

- "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: Define an interface for creating an object, but let subclasses decide which class to instantiate
- 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

Factory Patterns

Factory patterns are examples of creational patterns

Creational patterns abstract the object instantiation process.

They hide how objects are created and help make the overall system independent of how its objects are created and composed.

Class Factory patterns use inheritance to decide the object to be instantiated

Factory Method

Object Factory patterns delegate the instantiation to another object

Factory Object

Also called Abstract Factory

Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Use the Factory Method pattern in any of the following situations:

- A class can't anticipate the class of objects it must create
- A class wants its subclasses to specify the objects it creates
- Create objects without exposing the creation logic to the client
- Refer to newly created object using a common interface.

Factory Method

- Subtypes 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

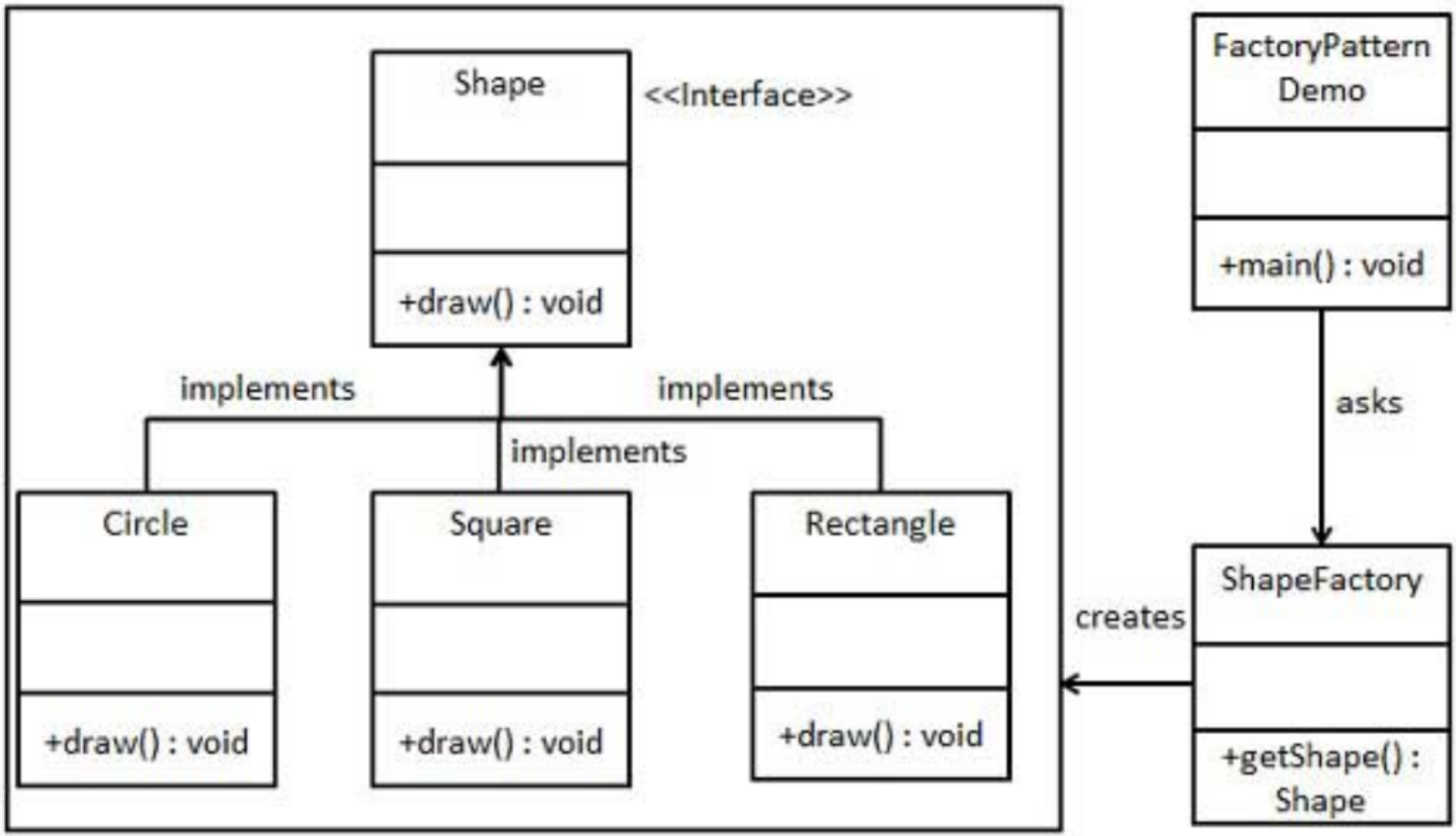
```
Matrix<Double> m = new DenseMatrix<Double>(100, 100);
```

Factory Instead

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

```
class MatrixFactory{
        public static Matrix createMatrix(...) {
                if (...)
                        return new SparseMatrix(...);
                else
                        return new DenseMatrix(...);
        }
}
Matrix m = MatrixFactory.createMatrix();
```

Example: A Shape Factory



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

```
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.
```

```
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
  }
}
```

```
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();
  }
}
```

Another 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

Factory Method

Where is the Factory Method?

createBicycle()

this.createBicycle calls appropriate method

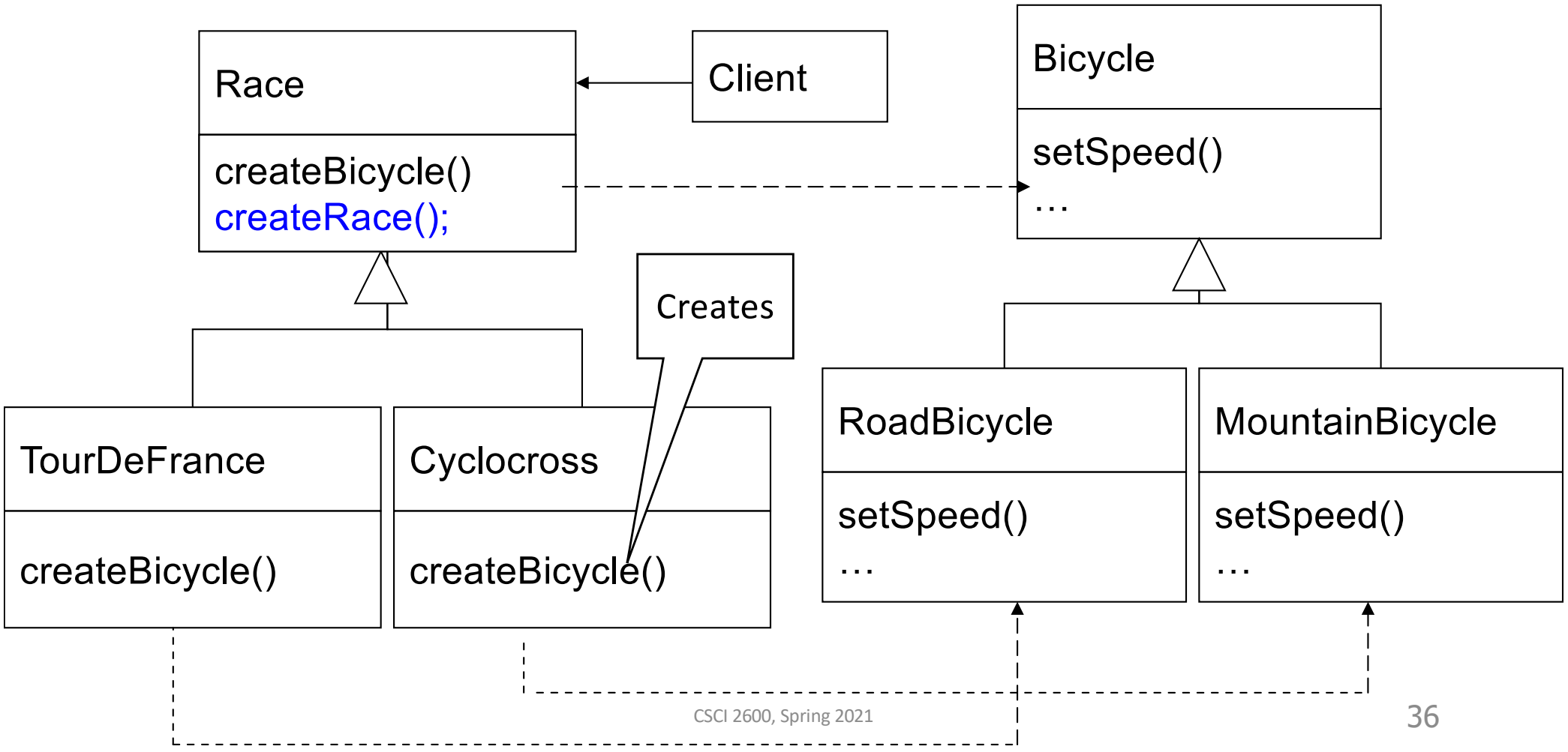
Race t = new TourDeFrance();
t.createRace(); // calls Race.createRace()

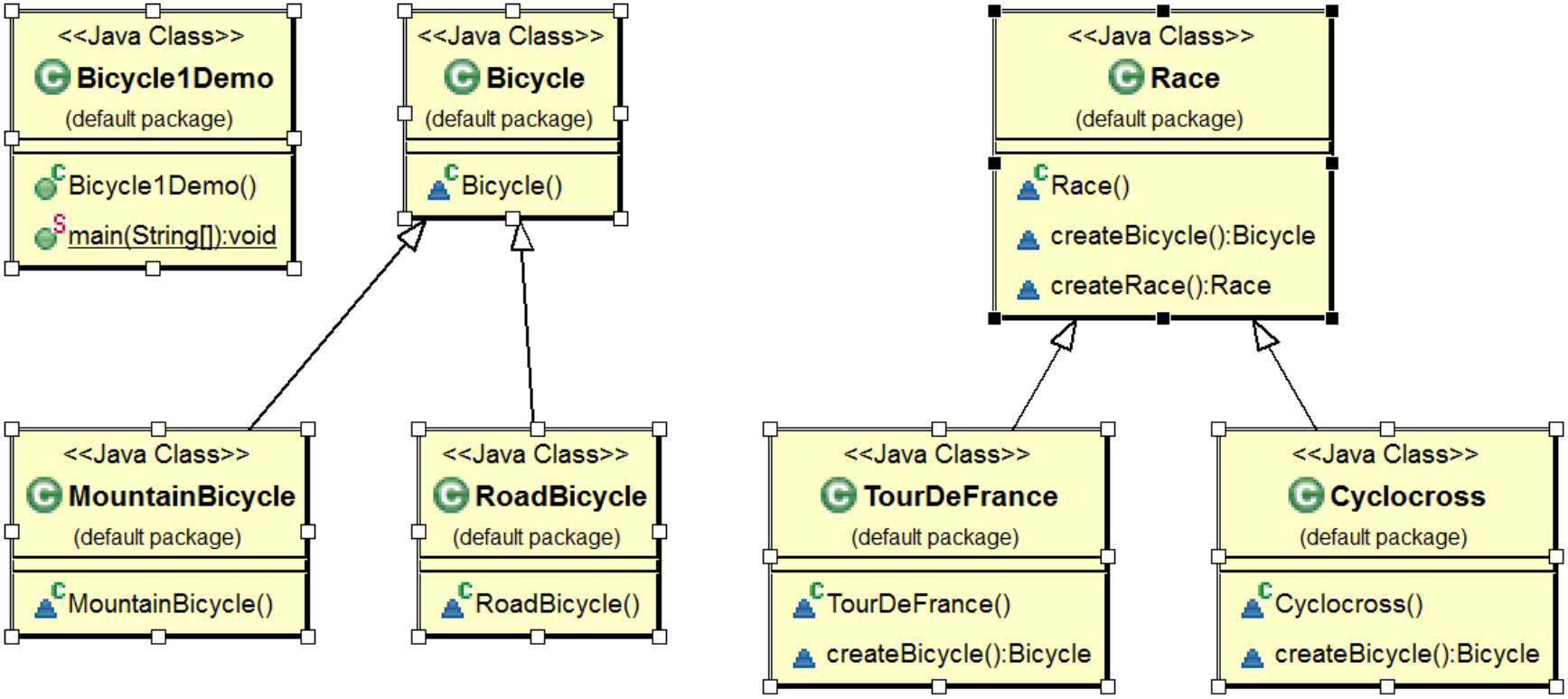
Race.createRace() calls this.createBicycle()

this is a TourDeFrance object, so TourDeFrance.createBicycle() is called
Returns a RoadBicycle

Parallel Hierarchies

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

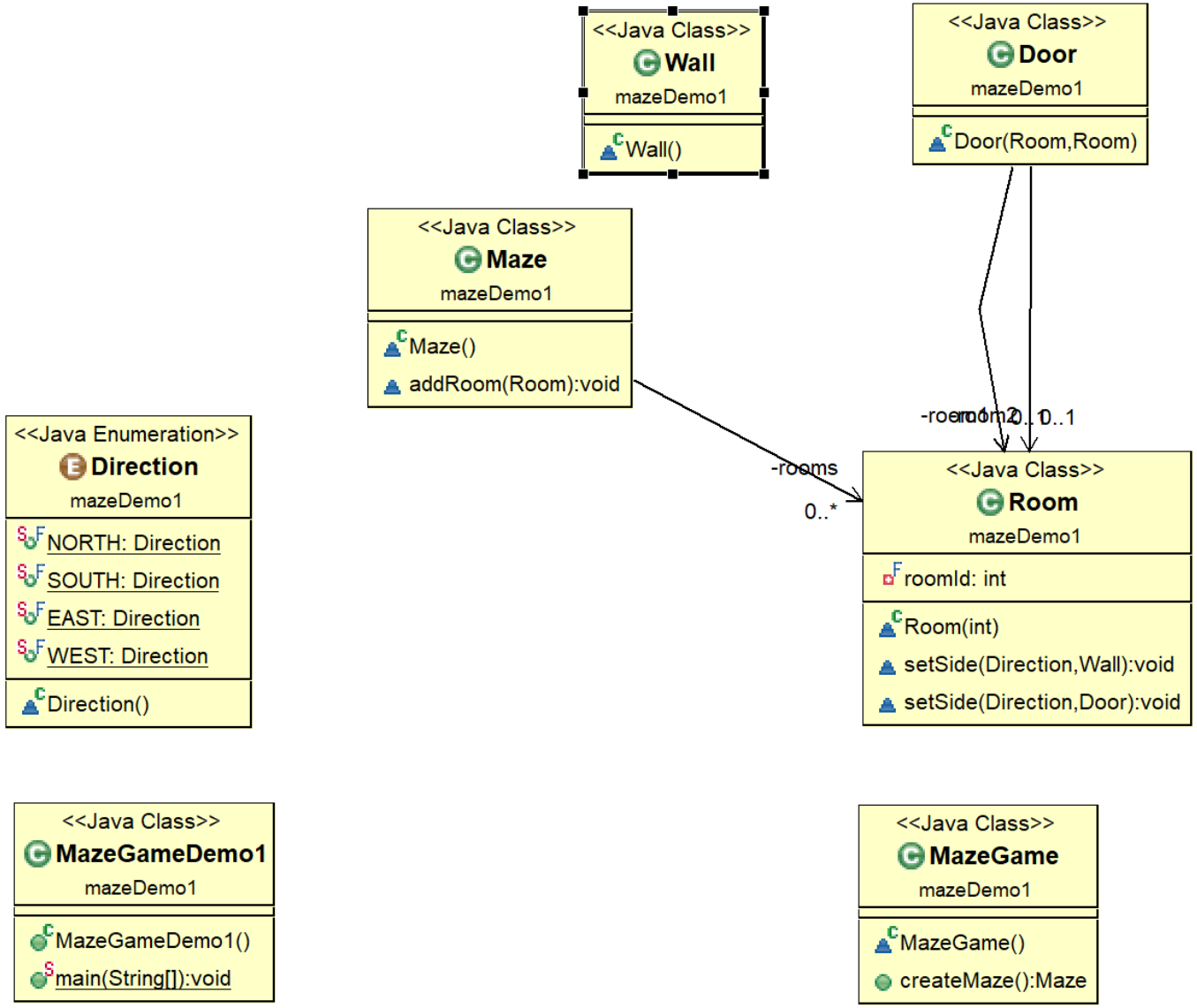



<<Java Class>>
Bicycle1Demo
(default package)
Bicycle1Demo()
main(String[]):void
<<Java Class>>
Bicycle
(default package)
Bicycle()
<<Java Class>>
Race
(default package)
Race()
createBicycle():Bicycle
createRace():Race
<<Java Class>>
MountainBicycle
(default package)
MountainBicycle()
<<Java Class>>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

Maze Game

```
public Maze createMaze() {
        Maze maze = new Maze();
        Room r1 = new Room(1);
        Room r2 = new Room(2);
        Door door = new Door(r1, r2);
        maze.addRoom(r1);
        maze.addRoom(r2);
        r1.setSide(Direction.NORTH, new Wall());
        …
        return maze;
}
```

Maze Game


<<Java Class>>
Wall
mazeDemo1
Wall()
<<Java Class>>
Door
mazeDemo1
Door(Room,Room)
<<Java Class>>
Maze
mazeDemo1
Maze()
addRoom(Room):void
-rooms
0..*
-room1
-room2
0..1
0..1
<<Java Class>>
Room
mazeDemo1
roomId: int
Room(int)
setSide(Direction,Wall):void
setSide(Direction,Door):void
<<Java Enumeration>>
Direction
mazeDemo1
NORTH: Direction
SOUTH: Direction
EAST: Direction
WEST: Direction
Direction()
<<Java Class>>
MazeGameDemo1
mazeDemo1
MazeGameDemo1()
main(String[]):void
<<Java Class>>
MazeGame
mazeDemo1
MazeGame()
createMaze():Maze

Maze Game

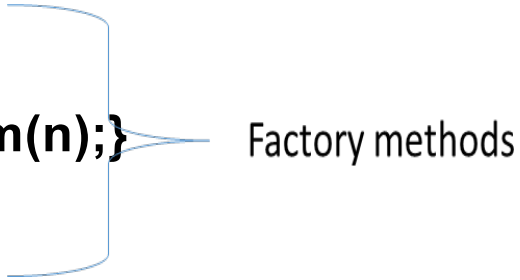
The problem with this createMaze() method is its inflexibility.

What if we wanted to have enchanted mazes with EnchantedRooms and EnchantedDoors?
Or a secret agent maze with DoorWithLock and WallWithHiddenDoor?

We would have to make significant edits

Maze Game version 2

```
class MazeGame {
// Create the maze.

public Maze makeMaze() {return new Maze();}
public Room makeRoom(int n) {return new Room(n);}
public Wall makeWall() {return new Wall();}
public Door makeDoor(Room r1, Room r2)
{return new Door(r1, r2);}

public Maze createMaze() {
    Maze maze = this.makeMaze();
    Room r1 = this.makeRoom(1);
    Room r2 = this.makeRoom(2);
    Door door = this.makeDoor(r1, r2);

// the rest is the same as before ...
    …
}
```



Maze Game version 2

We made createMaze() just slightly more complex, but a lot more flexible!

Consider this EnchantedMazeGame class:

```
public class EnchantedMazeGame extends MazeGame {
        public Room makeRoom(int n) {return new EnchantedRoom(n);}
        public Wall makeWall() {return new EnchantedWall();}
        public Door makeDoor(Room r1, Room r2)
                {return new EnchantedDoor(r1, r2);}
}
```

The createMaze() method of MazeGame is inherited by EnchantedMazeGame and can be used to create regular mazes or enchanted mazes without modification

Maze Game version 2

- We can extend MazeGame to make an EnchantedMazeGame
  - Override makeRoom() etc. Like we did with Bicycle to return specialized rooms, walls, etc.

Yet Another Factory Method Abstraction

```
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
```

Factories

- The Bicycle and Maze examples conform with the GoF definition of Factory method
  - Parallel hierarchy of Objects and their Products and each Object class creates its own product.
- The term “factory method” is used outside of this context.
- In general “Factory method” refers to any method that can return a Product (maybe a subtype), depending on the context.

Factory Object

Also know as **Abstract Factory**

The responsibility of object creation is delegated to a factory object

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

The Factory Object pattern is very similar to the Factory Method pattern.

> The main difference between the two is that with the Factory Object pattern, a class delegates the responsibility of object instantiation to another object via composition
>
> The Factory Method pattern uses inheritance and relies on a subclass to handle the desired object instantiation.

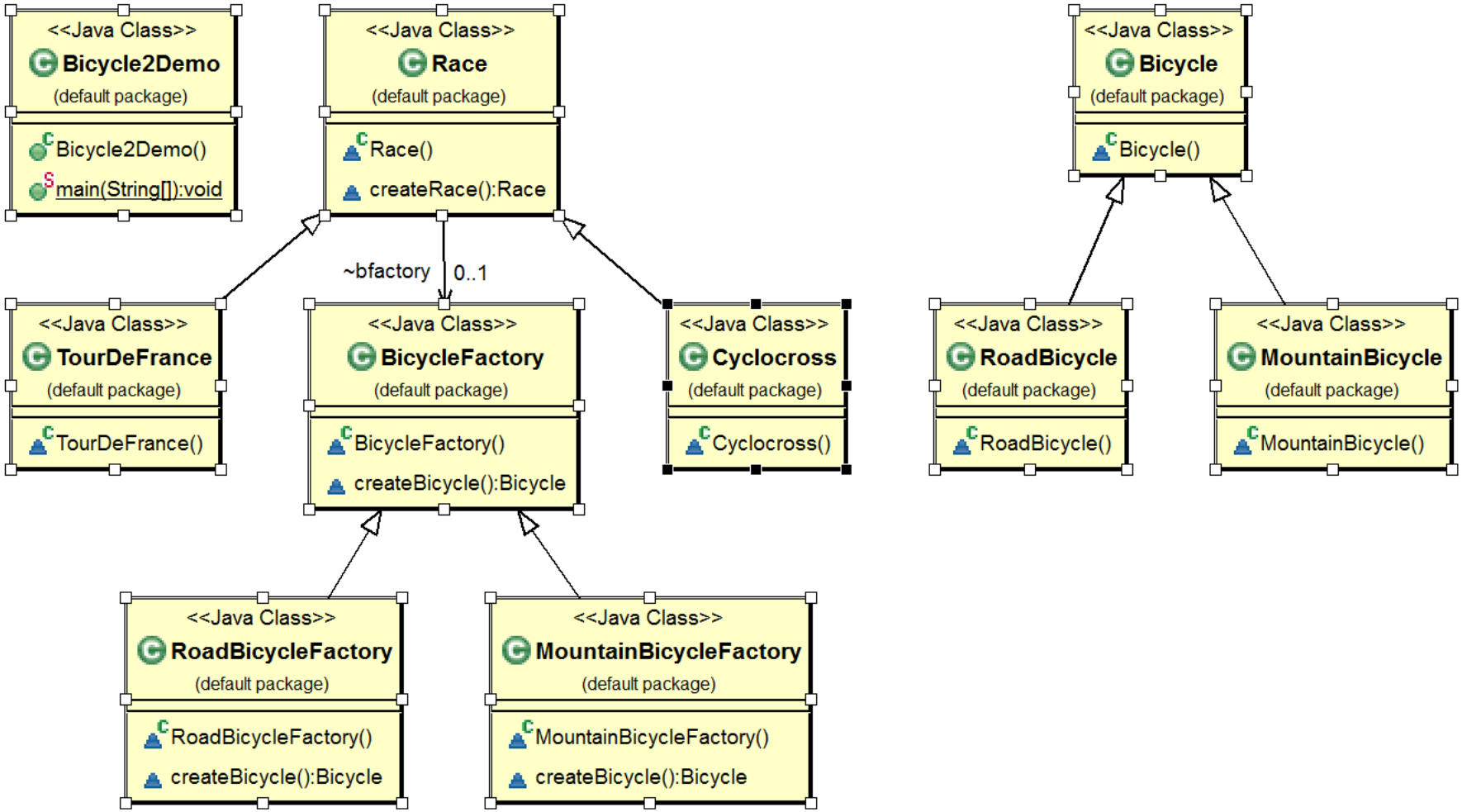
Actually, the delegated object frequently uses factory methods to perform the instantiation!

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>>
Bicycle
(default package)
Bicycle()
<<Java Class>>
TourDeFrance
(default package)
TourDeFrance()
<<Java Class>>
BicycleFactory
(default package)
BicycleFactory()
createBicycle():Bicycle
<<Java Class>>
Cyclocross
(default package)
Cyclocross()
<<Java Class>>
RoadBicycle
(default package)
RoadBicycle()
<<Java Class>>
MountainBicycle
(default package)
MountainBicycle()
<<Java Class>>
RoadBicycleFactory
(default package)
RoadBicycleFactory()
createBicycle():Bicycle
<<Java Class>>
MountainBicycleFactory
(default package)
MountainBicycleFactory()
createBicycle():Bicycle

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

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

Factory methods encapsulated into Factory classes

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());
```

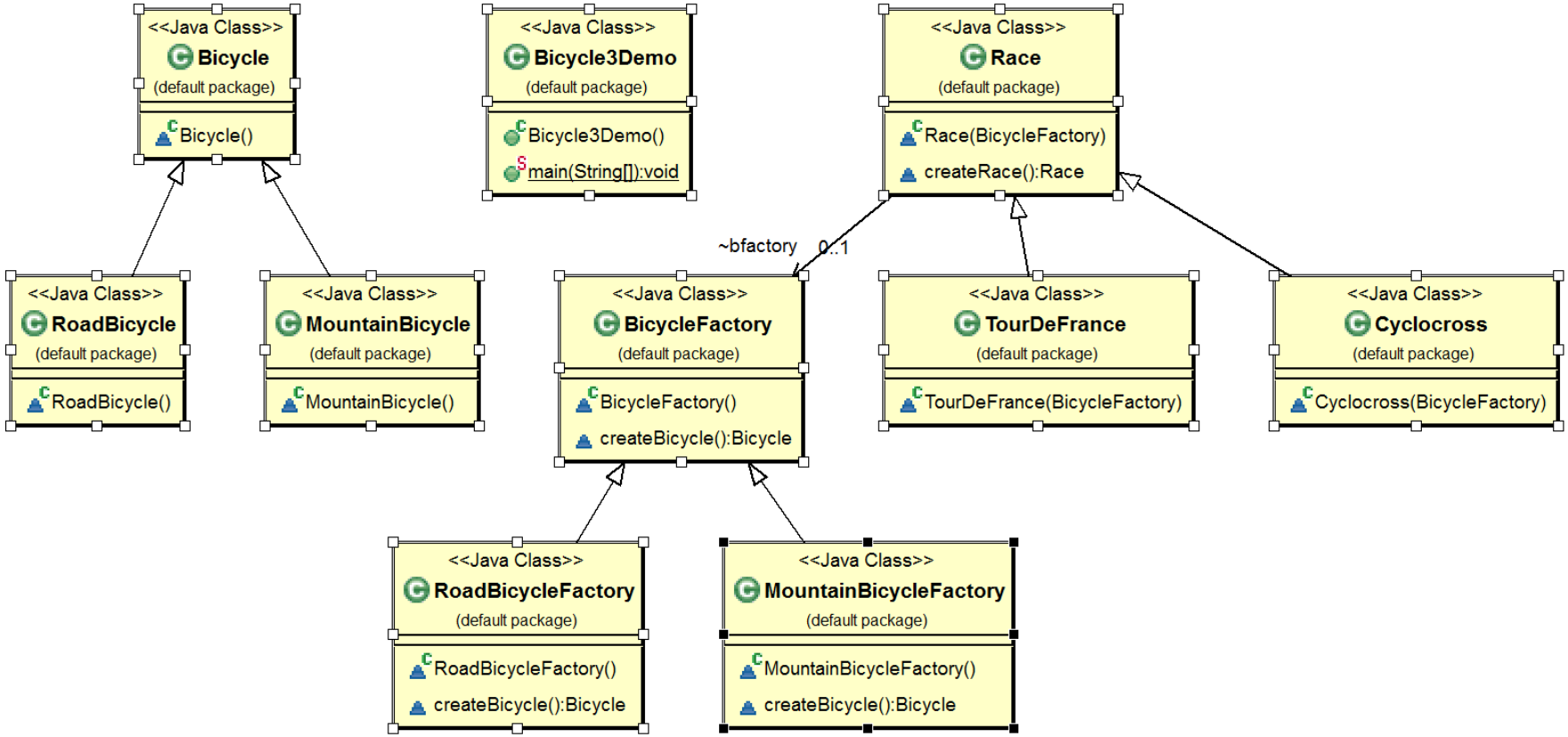
- To specify a different race/bicycle need only change one line:

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

or

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

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

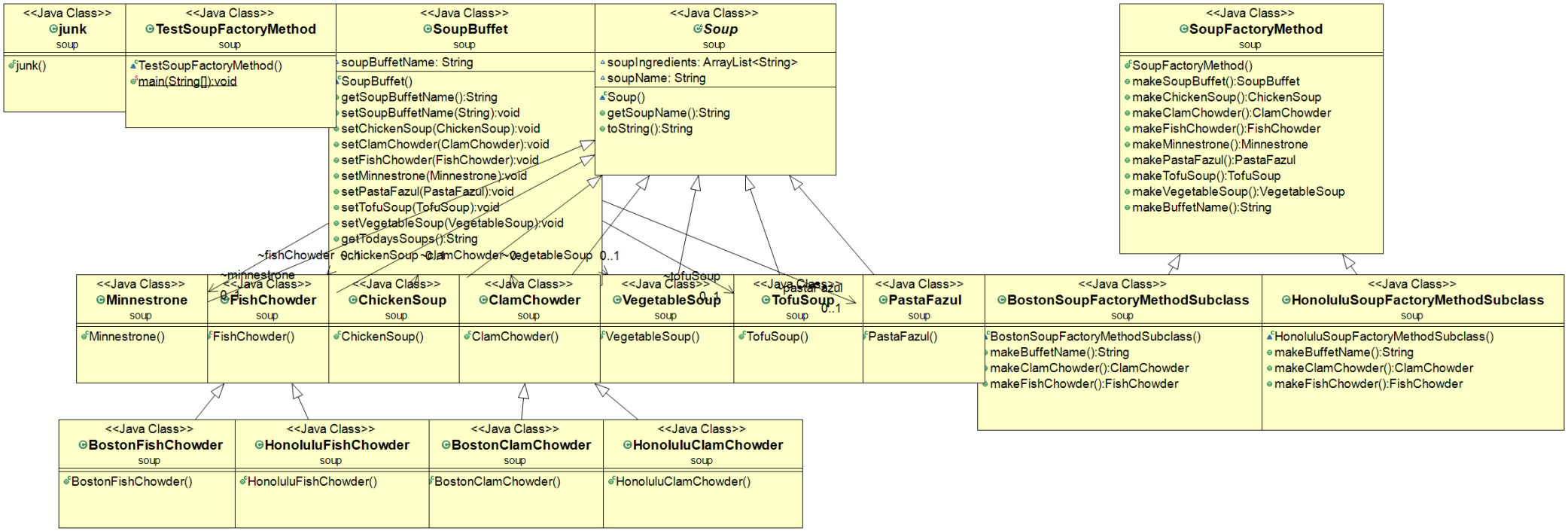

<<Java Class>>
Bicycle
(default package)
Bicycle()
<<Java Class>>
Bicycle3Demo
(default package)
Bicycle3Demo()
main(String[]):void
<<Java Class>>
Race
(default package)
Race(BicycleFactory)
createRace():Race
~bfactory
0..1
<<Java Class>>
RoadBicycle
(default package)
RoadBicycle()
<<Java Class>>
MountainBicycle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

Bicycle Factories

- In the Bicycle factory method, Race class is the factory
  - createRace() uses createBicycle() to instantiate bikes
  - Subclasses override createBicycle() to return appropriate bike type
- In Bicycle object factory, Race class contains a BicycleFactory Object
  - Race uses factory object to create bikes
  - Subclasses instantiate the appropriate BicycleFactory object
  - We can also pass the BicycleFactory object to the subclass constructors
    - For example, TourDeFrance could use MountainBicycleFactory or RoadBicycleFactory

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
~chickenSoup
~clamChowder
~vegetableSoup
0..1
~minnestrone
~tofuSoup
pastaFazul
<<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()

Factories 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);
```

- DateFormat.getDateInstance(); returns an object that can create the appropriate type of date, i.e. a factory.

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

Functional Factory Pattern

Replace factory class with a Supplier
function method that “supplies” an object

```
interface Bicycle {
        Bicycle getBike();
}

class MountainBicycle implements Bicycle {
        public Bicycle getBike() {
                System.out.println("Mountain Bicycle");
                return this;
        }
}

final Supplier<Bicycle> mountainBikeFactory = () -> {
        return new MountainBicycle();
};
```

To use it

```
Bicycle b1 = mountainBikeFactory.get().getBike();
```

Dependency Injection

- In Java, we can decide what **`Factory`** to initialize with at runtime
- An external file specifies a value for “BikeFactory”, factory in plain text, say “TricycleFactory”
- **`Re`**ad file and uses Java reflection to load and instantiate class, **`TricycleFactory`**

Dependency Injection

```
public static void main(String[] args) {
      try {
          File f = new File("Data/bikefactory");
          BufferedReader b = new BufferedReader(new FileReader(f));
          String factory = b.readLine();  // get the factory type from file
          if(factory != null) {
                // need to add package so forName can find the class
                Class factoryClazz = Class.forName(factory);
                BicycleFactory bfactory = (BicycleFactory) factoryClazz.newInstance();
                Race t = new TourDeFrance(bfactory); // we could change bike type here
                t.createRace();
          }
       } catch (IOException | ClassNotFoundException e) {
                e.printStackTrace();
       } catch (InstantiationException e) {
                e.printStackTrace();
       } catch (IllegalAccessException e) {
                e.printStackTrace();
     }
}
```

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`** class declares **`protected Object clone()`**

Prototype

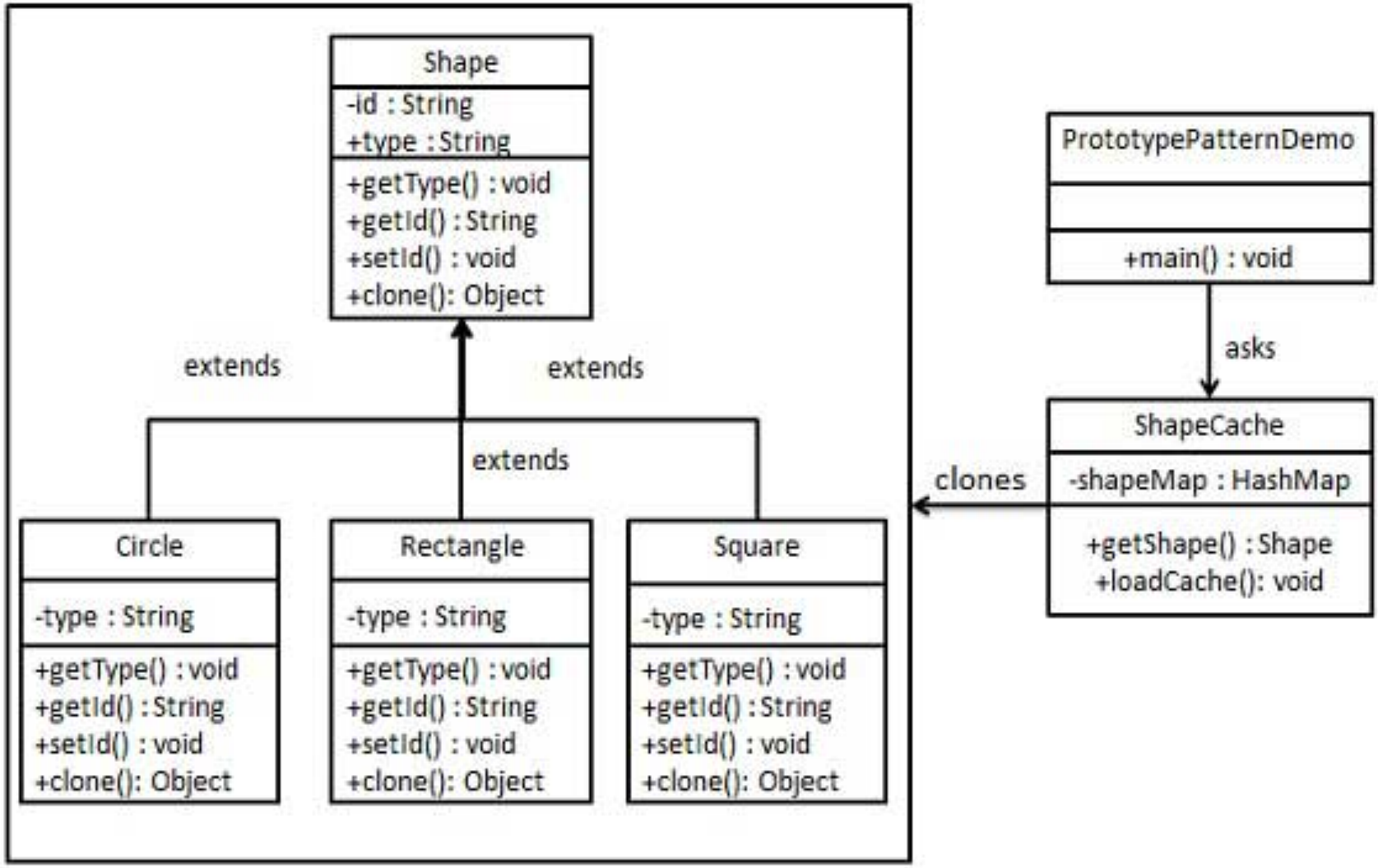
- Prototype interface creates a clone of the current object.
- Useful when creation of the object directly is costly.
  - For example, an object is to be created after a costly database operation
  - Cache the object
    - return its clone on next request
    - only update the database when needed
 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

Java Clone

- Dictionary: “make an identical copy of“.
- By default, Java cloning is “field by field copy”
  - Object class has clone() method
  - Object class does not have idea about the structure of class on which clone() method will be invoked.
  - JVM when called for cloning, does the following things:
    - If the class has only primitive data type members then a completely new copy of the object will be created and the reference to the new object copy will be returned.
    - If the class contains members of any class type then only the object references to those members are copied
      - The member references in both the original object as well as the cloned object refer to the same object.
- You can always override this behavior and specify your own. This is done by overriding the clone() method.

Java Clone

- If a class needs to support cloning it has to do following things:
  - You must implement Cloneable interface.
  - You must override clone() method from Object class.
  - clone() method is not in Cloneable interface
- A class implements the Cloneable interface to indicate to the Object.clone() method that it is legal for that method to make a field-for-field copy of instances of that class.
- Invoking Object's clone method on an instance that does not implement the Cloneable interface results in the exception *CloneNotSupportedException* being thrown.



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

```
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();  // call Object.clone()
    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
    }

    return clone;
  }
}
```

```
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
```

```
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();  // return a clone of the object
  }

  // 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());
```

Are clones dangerous?

- Object.clone() performs field (variable) by field copy
- If fields are references, reference is copied
  - The fields in the original and the clone refer to the same objects
- If the class allows changes to a referenced object, it is changed for both objects.
  - Same problem for Lists, Maps, etc.
  - If you make a copy, only references are copied
  - If the references refer to mutable objects, they can be changed – possible rep exposure
- To avoid this make a deep copy
  - Make a clone of each referenced object recursively
  - Watch out for sets or arrays of mutable objects

Sharing

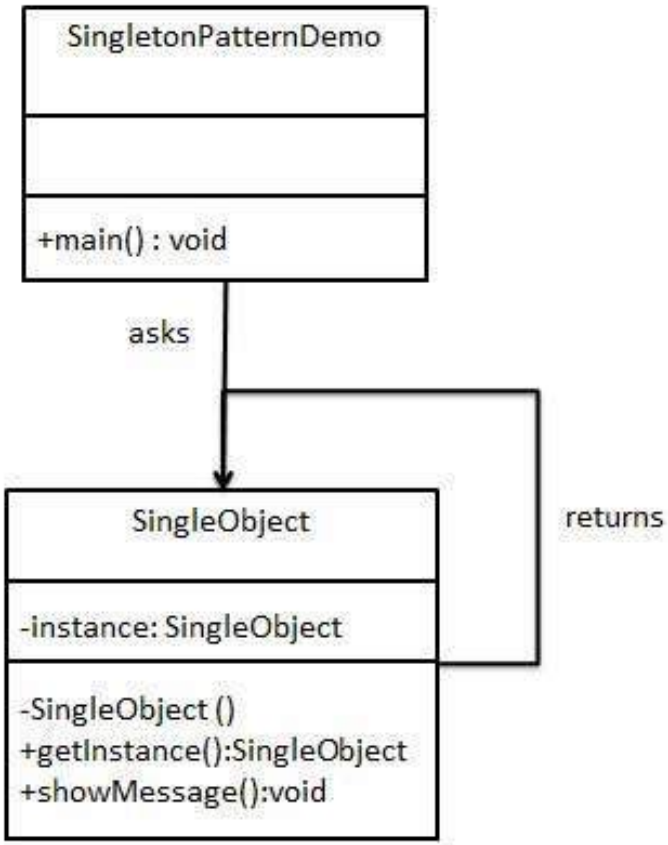
- Recall that constructors always create a new object, never a pre-existing one
- In many situations, we would like a pre-existing object
- Singleton pattern: only one object ever exists
  - A factory object is almost always a singleton
  - Device drivers, music players, database interface
- Interning pattern: only one object with a given abstract value exists

Singleton Pattern

- Motivation: there must be a single 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
}
```

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



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

```
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 from the ouside
   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

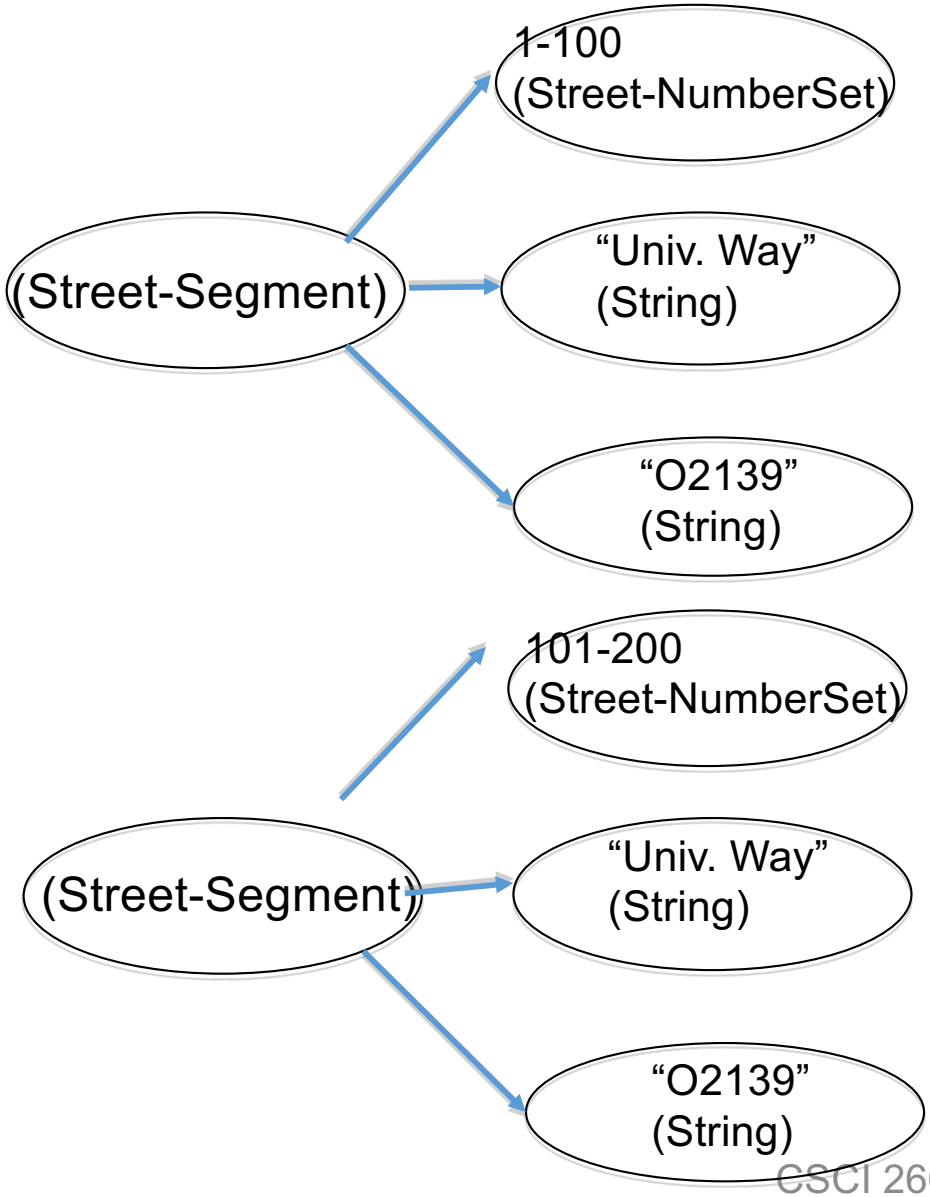
Static initializer --- executed when class is loaded

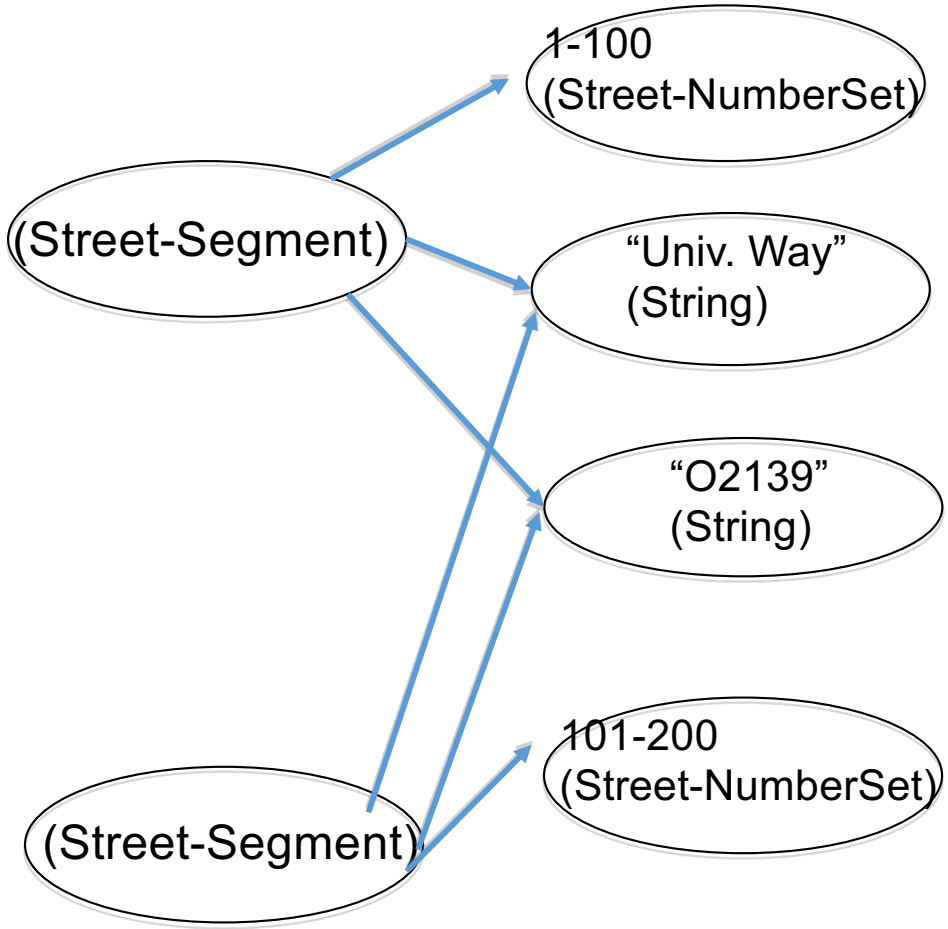
```
public class UserDatabaseSource
        implements UserDatabase {
  private static UserDatabase theInstance =
        new UserDatabaseSource();
  private UserDatabaseSource() { … }
  public static UserDatabase getInstance() {
     return theInstance; }

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

Interning Pattern

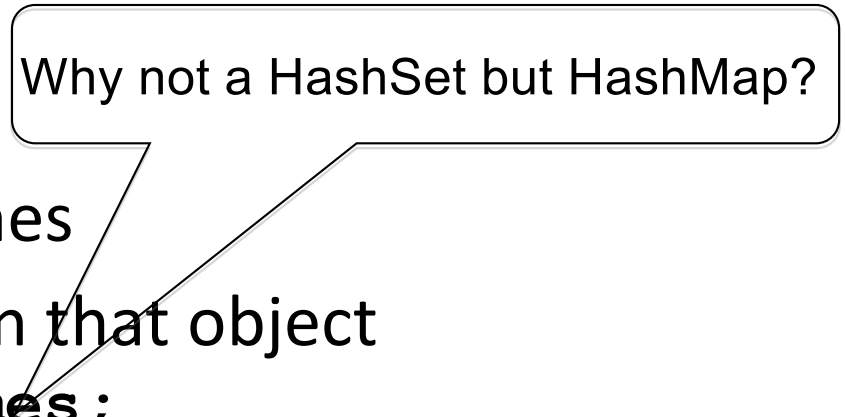
- Not a GoF design pattern
- Reuse existing object with same value, instead of creating a new one
  - E.g., why create multiple Strings “car”? Create a single instance of String “car”!
  - Less space
  - May compare with == instead of **`equals`** and speed the program up
- Interning is applied to immutable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

Interning Pattern





Interning Pattern



- 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?

- 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.
- Deprecated since Java 9

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.