CSE 331

Design Patterns 1:
Iterator, Adapter, Singleton, Flyweight

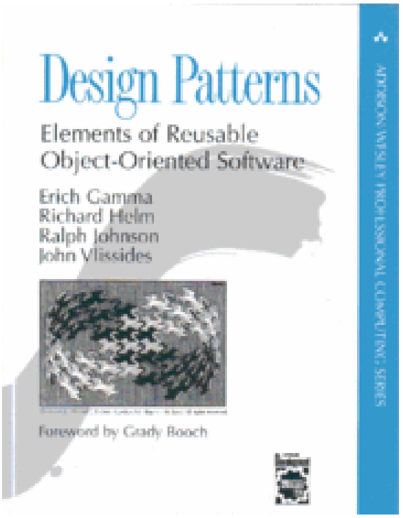
slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer

http://www.cs.washington.edu/331/

Design patterns

- **design pattern**:
  A standard solution to a common software problem in a context.
  - describes a recurring software structure or idiom
  - is abstract from any particular programming language
  - identifies classes and their roles in the solution to a problem

- in 1990 a group called the *Gang of Four* or "GoF" (Gamma, Helm, Johnson, Vlissides) compile a catalog of design patterns
  - 1995 book *Design Patterns: Elements of Reusable Object-Oriented Software* is a classic of the field



Benefits of using patterns

- Patterns give a design **common vocabulary** for software design:
  - Allows engineers to abstract a problem and talk about that abstraction in isolation from its implementation.
  - A culture; domain-specific patterns increase design speed.
- **Capture expertise** and allow it to be communicated:
  - Promotes design reuse and avoid mistakes.
  - Makes it easier for other developers to understand a system.
- **Improve documentation** (less is needed):
  - Improve understandability (patterns are described well, once).

Gang of Four (GoF) patterns

- **Creational Patterns** *(abstracting the object-instantiation process)*
  - Factory Method Abstract Factory Singleton
  - Builder Prototype
- **Structural Patterns** *(how objects/classes can be combined)*
  - Adapter Bridge Composite
  - Decorator Facade Flyweight
  - Proxy
- **Behavioral Patterns** *(communication between objects)*
  - Command Interpreter Iterator
  - Mediator Observer State
  - Strategy Chain of Responsibility Visitor
  - Template Method

Describing a pattern

- *Problem:* In what situation should this pattern be used?
- *Solution:* What should you do? What is the pattern?
  - describe details of the objects/classes/structure needed
  - should be somewhat language-neutral
- *Advantages:* Why is this pattern useful?
- *Disadvantages:* Why might someone not want this pattern?

Pattern: Iterator

objects that traverse collections

Iterator pattern

- *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:*
  - Provide a standard *iterator* object supplied by all data structures.
  - The implementation performs traversals, does bookkeeping.
    - The implementation has knowledge about the representation.
  - Results are communicated to clients via a standard interface.
- *Disadvantages:*
  - Iteration order is fixed by the implementation, not the client.
  - Missing various potentially useful operations (add, set, etc.).

Pattern: Adapter

an object that fits another object into a given interface

Adapter pattern

- *Problem:* We have an object that contains the functionality we need, but not in the way we want to use it.
  - Cumbersome / unpleasant to use. Prone to bugs.

- *Example:*
  - We are given an Iterator, but not the collection it came from.
  - We want to do a for-each loop over the elements, but you can't do this with an Iterator, only an `Iterable`:

```
public void printAll(Iterator<String> itr) {
    // error: must implement Iterable
    for (String s : itr) {
        System.out.println(s);
}   }
```

Adapter in action

- *Solution:* Create an **adapter object** that bridges the provided and desired functionality.

```
public class IterableAdapter implements Iterable<String> {
    private Iterator<String> iterator;

    public IterableAdapter(Iterator<String> itr) {
        this.iterator = itr;
    }

    public Iterator<String> iterator() {
        return iterator;
    }
}
...
public void printAll(Iterator<String> itr) {
    IterableAdapter adapter = new IterableAdapter(itr);
    for (String s : adapter) { ... }  // works
}
```

Pattern: Singleton

A class that has only a single instance

Creational Patterns

- Constructors in Java are inflexible:
  - Can't return a subtype of the class they belong to.
  - Always returns a fresh new object; can never re-use one.
- Creational factories:
  - Factory method
  - Abstract Factory object
  - Prototype
  - Dependency injection
- Sharing:
  - Singleton
  - Interning
  - Flyweight

Restricting object creation

- *Problem:* Sometimes we really only ever need (or want) one instance of a particular class.
  - Examples: keyboard reader, bank data collection, game, UI
  - We'd like to make it illegal to have more than one.

- *Issues:*
  - Creating lots of objects can take a lot of time.
  - Extra objects take up memory.
  - It is a pain to deal with different objects floating around if they are essentially the same.
  - Multiple objects of a type intended to be unique can lead to bugs.
    - What happens if we have more than one game UI, or account manager?

Singleton pattern

- **singleton**: An object that is the only object of its type.
  (one of the most known / popular design patterns)

  - Ensuring that a class has at most one instance.
  - Providing a global access point to that instance.
    - e.g. Provide an accessor method that allows users to see the instance.

- *Benefits:*
  - Takes responsibility of managing that instance away from the programmer (illegal to construct more instances).
  - Saves memory.
  - Avoids bugs arising from multiple instances.

Restricting objects

- One way to avoid creating objects: use static methods
  - Examples: `Math`, `System`
  - Is this a good alternative choice? Why or why not?

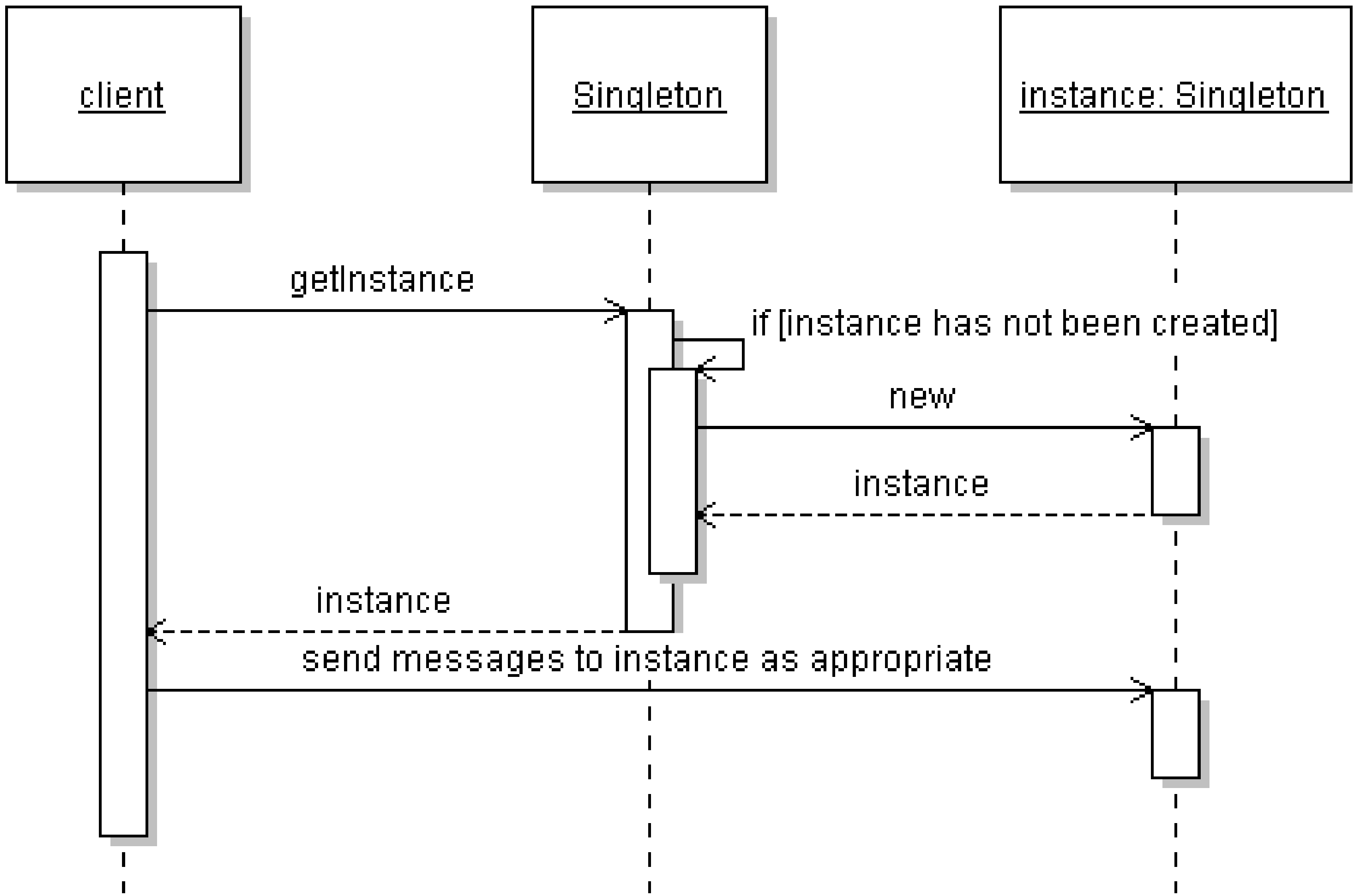
- *Disadvantage*: Lacks flexibility.
  - Static methods can't be passed as an argument, nor returned.

- *Disadvantage*: Cannot be extended.
  - Example: Static methods can't be subclassed and overridden like an object's methods could be.

Implementing Singleton

- Make constructor(s) `private` so that they can not be called from outside by clients.
- Declare a single `private static` instance of the class.
- Write a public `getInstance()` or similar method that allows access to the single instance.
  - May need to protect / synchronize this method to ensure that it will work in a multi-threaded program.

Singleton sequence diagram



Singleton example

- Class `RandomGenerator` generates random numbers.

```
public class RandomGenerator {
    private static final RandomGenerator gen =
            new RandomGenerator();

    public static RandomGenerator getInstance() {
        return gen;
    }

    private RandomGenerator() {}

    ...
}
```

Lazy initialization

- Can wait until client asks for the instance to create it:

```
public class RandomGenerator {
    private static RandomGenerator gen = null;

    public static RandomGenerator getInstance() {
        if (gen == null) {
            gen = new RandomGenerator();
        }
        return gen;
    }

    private RandomGenerator() {}

    ...
}
```

Singleton Comparator

- Comparators make great singletons because they have no state:

```
public class LengthComparator
        implements Comparator<String> {
    private static LengthComparator comp = null;

    public static LengthComparator getInstance() {
        if (comp == null) {
            comp = new LengthComparator();
        }
        return comp;
    }

    private LengthComparator() {}

    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

Pattern: Flyweight

a class that has only one instance for each unique state

Redundant objects

- *Problem*: Redundant objects can bog down the system.
  - Many objects have the same state.

  - example: `File` objects that represent the same file on disk
    - `new File("mobydick.txt")`
    - `new File("mobydick.txt")`
    - `new File("mobydick.txt")`

      …
    - `new File("notes.txt")`

  - example: `Date` objects that represent the same date of the year
    - `new Date(4, 18)`
    - `new Date(4, 18)`

Flyweight pattern

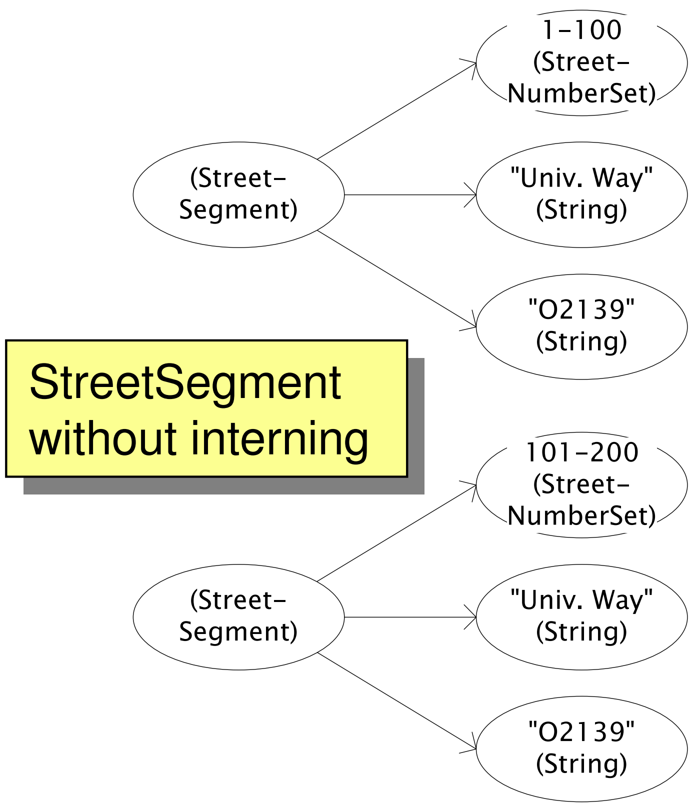
- **flyweight**: An assurance that no more than one instance of a class will have identical state.
  - Achieved by caching identical instances of objects.
  - Similar to singleton, but one instance for each unique object state.

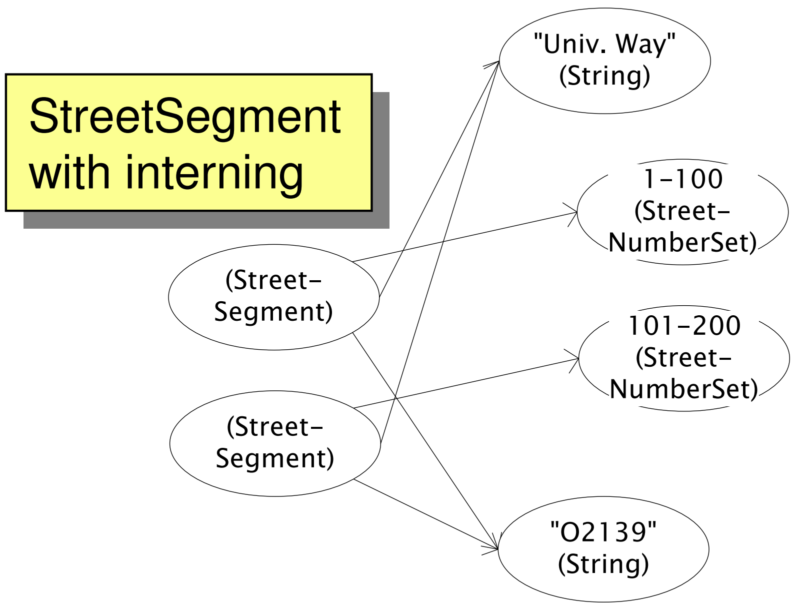
  - Useful when there are many instances, but many are equivalent.
  - Can be used in conjunction with Factory Method pattern to create a very efficient object-builder.

  - Examples in Java: `String`, `Image`, `Toolkit`, `Formatter`, `Calendar`, `JDBC`

Flyweight diagram

- Flyweighting shares objects and/or shares their internal state
  - saves memory
  - allows comparisons with == rather than `equals` (why?)





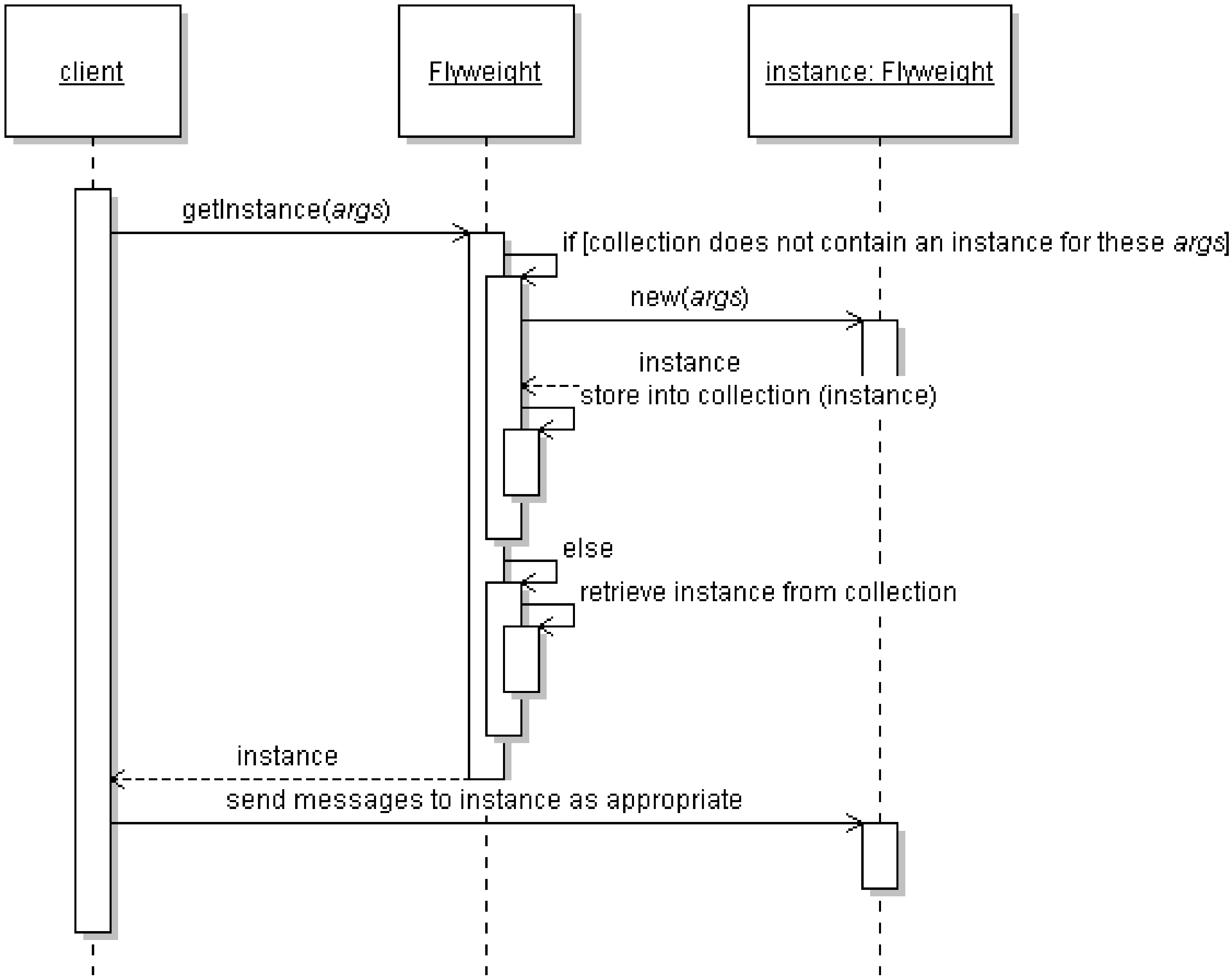
Implementing a Flyweight

- Flyweighting works best on *immutable* objects. (Why?)

- Class pseudo-code sketch:

```
public class Name {
  • static collection of instances
  • private constructor
  • static method to get an instance:
    if (we have created this kind of instance before):
        get it from the collection and return it.
    else:
        create a new instance, store it in the collection and return it.
}
```

Flyweight sequence diagram



Implementing a Flyweight

```
public class Flyweighted {
    private static Map<KeyType, Flyweighted> instances
            = new HashMap<KeyType, Flyweighted>();

    private Flyweighted(...) { ... }

    public static Flyweighted getInstance(KeyType key) {
        if (!instances.contains(key)) {
            instances.put(key, new Flyweighted(key));
        }
        return instances.get(key);
    }
}
```

Class before flyweighting

```
public class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }

    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}
```

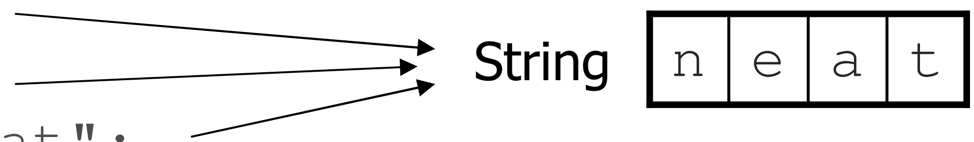
Class after flyweighting

```
public class Point {
    private static Map<String, Point> instances =
        new HashMap<String, Point>();

    public static Point getInstance(int x, int y) {
        String key = x + ", " + y;
        if (!instances.containsKey(key)) {
            instances.put(key, new Point(x, y));
        }
        return instances.get(key);
    }

    private final int x, y;  // immutable

    private Point(int x, int y) {
        ...
```

String flyweighting

- **interning**: Synonym for flyweighting; sharing identical instances.
  - Java `String` objects are automatically interned (flyweighted) by the compiler whenever possible.
  - If you declare two string variables that point to the same literal.
  - If you concatenate two string literals to match another literal.

```
String a = "neat";
String b = "neat";
String c = "n" + "eat";
```



- So why doesn't == always work with `String`s?

Limits of String flyweight

```
String a = "neat";
Scanner console = new Scanner(System.in);
String b = console.next();  // user types "neat"
if (a == b) { ...           // false
```

- There are many cases the compiler doesn't / can't flyweight:
  - When you build a string later out of arbitrary variables
  - When you read a string from a file or stream (e.g. `Scanner`)
  - When you build a new string from a `StringBuilder`
  - When you explicitly ask for a `new String` (bypasses flyweighting)
- You can force Java to flyweight a particular string with `intern`:

```
b = b.intern();
if (a == b) { ...           // true
```

String interning questions

```
String fly  = "fly";   String weight  = "weight";
String fly2 = "fly";   String weight2 = "weight";
```

- Which of the following expressions are `true`?
  a) `fly == fly2`
  b) `weight == weight2`
  c) `"fly" + "weight" == "flyweight"`
  d) `fly + weight == "flyweight"`

  ```
  String flyweight = new String("fly" + "weight");
  ```

  e) `flyweight == "flyweight"`

  ```
  String interned1 = (fly + weight).intern();
  String interned2 = flyweight.intern();
  ```

  f) `interned1 == "flyweight"`
  g) `interned2 == "flyweight"`