# What Is An Object-Oriented Language?

- [Cardelli/Wegner 1985, §1.3]

  "... it is useful to define object-oriented languages as extensions of procedure-oriented languages that support typed data abstractions with inheritance. Thus we say that a language is object-oriented iff it satisfies the following requirements:

  - It supports objects that are data abstractions with an interface of named operations and a hidden local state[.]
  - Objects have an associated object type[.]
  - Types may inherit attributes from supertypes[.]"

# What Is An Object-Oriented Language?

- [Cardelli/Wegner 1985, §1.3]

  "… it is useful to define object-oriented languages as extensions of procedure-oriented languages that support typed data abstractions with inheritance. Thus we say that a language is object-oriented iff it satisfies the following requirements:

  - It supports objects that are data abstractions with an interface of named operations and a hidden local state[.]
  - Objects have an associated object type[.]
  - Types may inherit attributes from supertypes[.]"

- [Cooper/Torczon 2004, §6.3.3]

  "Fundamentally, object orientation is a reorganization of the program's name space from a procedure-oriented scheme to a data-oriented scheme.[...] [O]bject-oriented languages differ from procedural languages in that they need some additional compile-time and run-time support. [...] An object is an abstraction that has one or more internal members. These members can be data items, code that manipulates those data items, or other objects. [...] A class is an abstraction that groups together similar objects."

# Name Spaces, Take 1: “Shared-Nothing”

```
extern double L2_dist(double x, double y);
extern double L1_dist(double x, double y);

typedef struct {
    int x, y;
    double (*dist)(double, double);
} Point;

int main(void) {
    Point p1 = {10, 20, L2_dist};
    Point p2 = {30, 40, L1_dist};
    return 0;
}
```

# Name Spaces, Take 1: “Shared-Nothing”

- Only prototypes.
- Greatest flexibility, but wasteful and tedious.

```
extern double L2_dist(double x, double y);
extern double L1_dist(double x, double y);

typedef struct {
    int x, y;
    double (*dist)(double, double);
} Point;

int main(void) {
    Point p1 = {10, 20, L2_dist};
    Point p2 = {30, 40, L1_dist};
    return 0;
}
```

# Name Spaces, Take 2: Code Reuse

```
extern double L2_dist(double x, double y);

typedef struct {
    int x, y;
    double (*dist)(double, double);
} Point;

int main(void) {
    Point p1 = {10, 20, L2_dist};
    Point p2 = {30, 40, L2_dist};
    return 0;
}
```

# Name Spaces, Take 2: Code Reuse

- Greater uniformity.
- Still no grouping construct for similar objects.

```
extern double L2_dist(double x, double y);

typedef struct {
    int x, y;
    double (*dist)(double, double);
} Point;

int main(void) {
    Point p1 = {10, 20, L2_dist};
    Point p2 = {30, 40, L2_dist};
    return 0;
}
```

# Name Spaces, Take 3: Code Reuse+Group

```
extern double L2_dist(double x, double y);

typedef struct {
    int count;
    double (*dist)(double, double);
} _PointClass;

typedef struct {
    _PointClass *cp;
    int x, y;
} Point;

int main(void) {
    _PointClass _pc = {0, L2_dist};
    Point p1 = {_pc, 10, 20};
    Point_p2 = _pc, 30, 40};
    return 0;
}
```

# Name Spaces, Take 3: Code Reuse+Group

- The data members of `Point` (aka instance variables) are unique for each instance.

```
extern double L2_dist(double x, double y);

typedef struct {
    int count;
    double (*dist)(double, double);
} _PointClass;

typedef struct {
    _PointClass *cp;
    int x, y;
} Point;

int main(void) {
    _PointClass _pc = {0, L2_dist};
    Point p1 = {_pc, 10, 20};
    Point_p2 = _pc, 30, 40};
    return 0;
}
```

# Name Spaces, Take 3: Code Reuse+Group

- The data members of `Point` (aka instance variables) are unique for each instance.
- The code members (aka instance methods) are consolidated into a `_PointClass` object and are shared among `Point` objects.
  - The object record of `Point` has changed.
  - Accessing `dist` requires an extra level of indirection.

```
extern double L2_dist(double x, double y);

typedef struct {
    int count;
    double (*dist)(double, double);
} _PointClass;

typedef struct {
    _PointClass *cp;
    int x, y;
} Point;

int main(void) {
    _PointClass _pc = {0, L2_dist};
    Point p1 = {_pc, 10, 20};
    Point_p2 = _pc, 30, 40};
    return 0;
}
```

# Name Spaces, Take 3: Code Reuse+Group

- The data members of `Point` (aka instance variables) are unique for each instance.
- The code members (aka instance methods) are consolidated into a `_PointClass` object and are shared among `Point` objects.
  - The object record of `Point` has changed.
  - Accessing `dist` requires an extra level of indirection.
- A class is an object that describes the properties of other objects.
  - A class can have its own class variables and class methods.

```
extern double L2_dist(double x, double y);

typedef struct {
    int count;
    double (*dist)(double, double);
} _PointClass;

typedef struct {
    _PointClass *cp;
    int x, y;
} Point;

int main(void) {
    _PointClass _pc = {0, L2_dist};
    Point p1 = {_pc, 10, 20};
    Point_p2 = _pc, 30, 40};
    return 0;
}
```