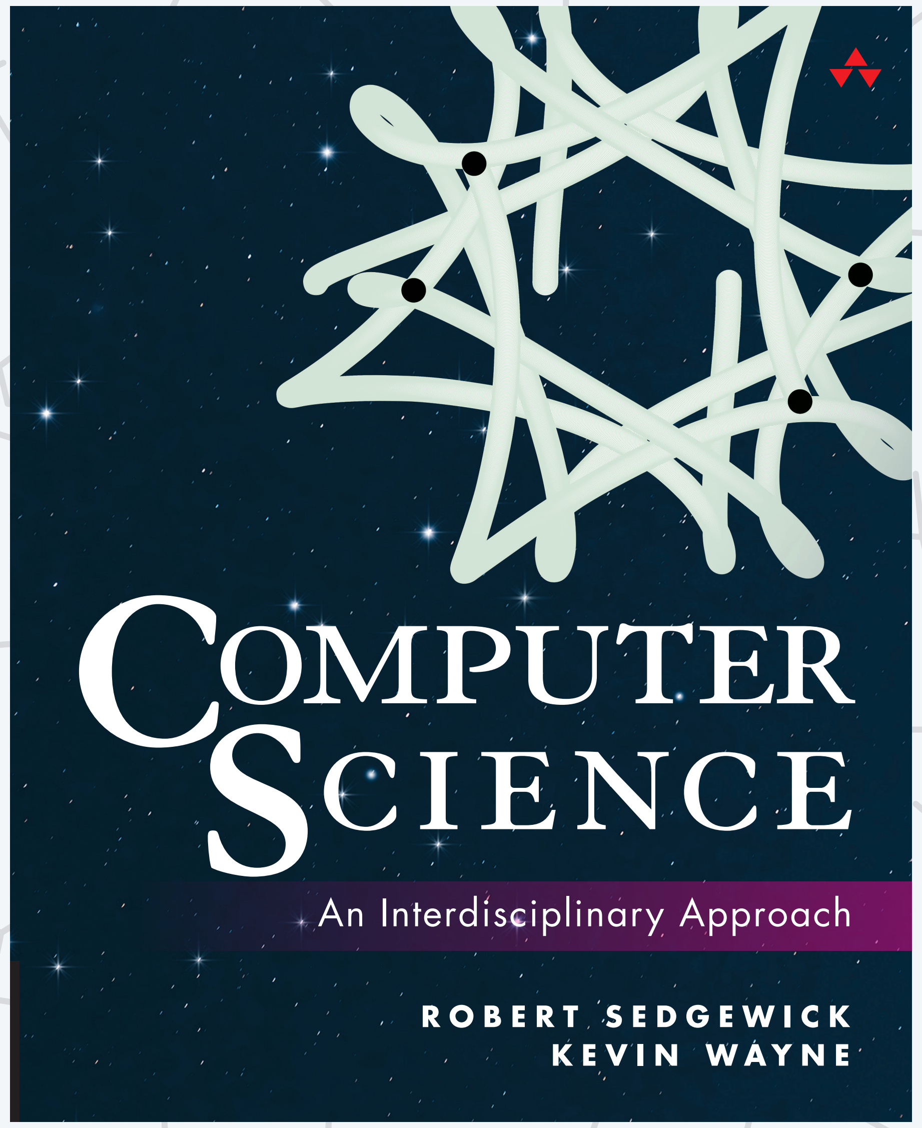



https://introcs.cs.princeton.edu

# 3.3 Designing Data Types

- *encapsulation*
- *immutability*
- *static variables and methods*
- *exceptions*
- *special references*
- *spatial vectors*

# Objects

Data type. A set of values and a set of operations on those values.

Java class. Java's mechanism for defining a new data type.

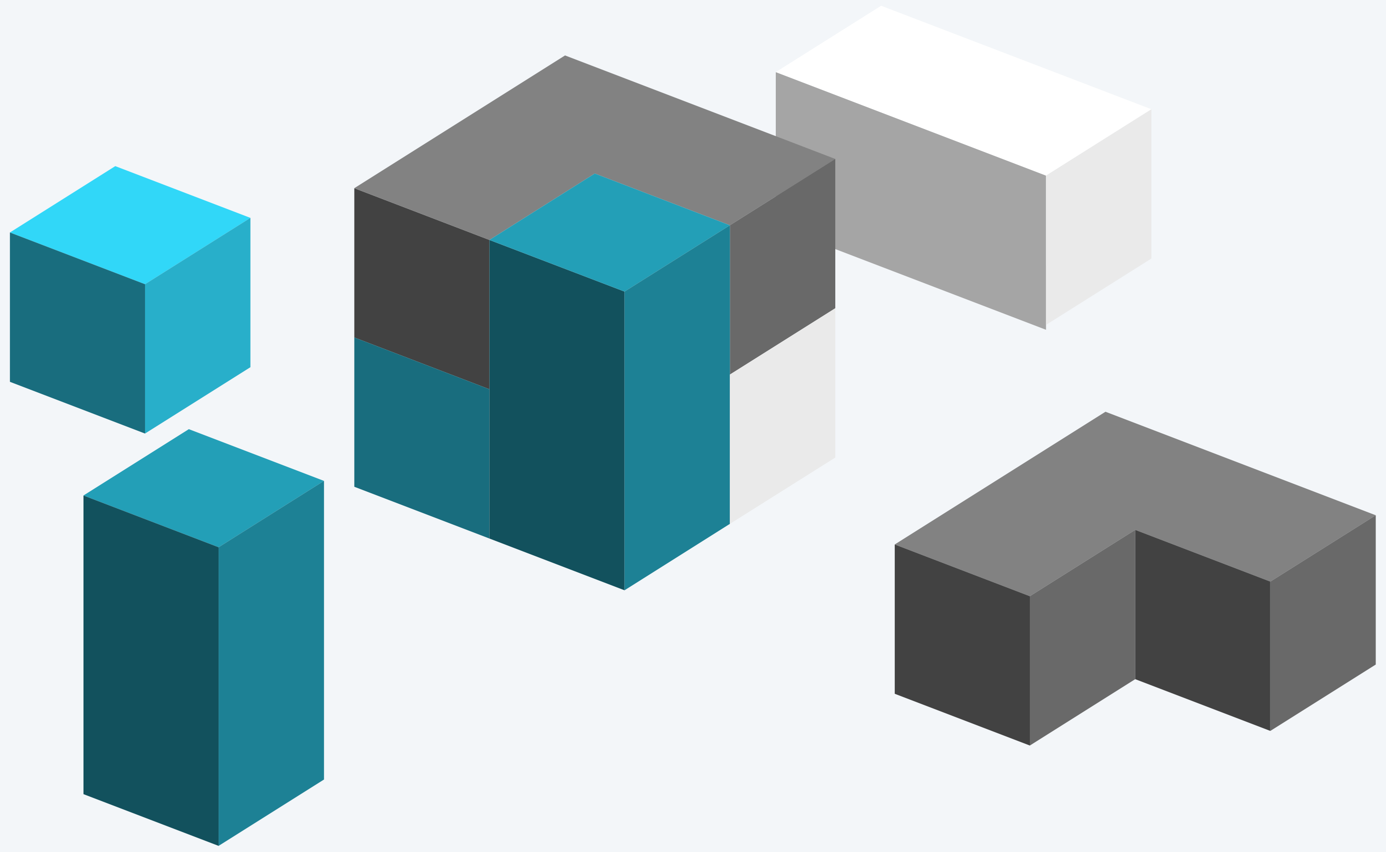
Object. An instance of a data type that has

- State: value from its data type.
- Behavior: actions defined by the data type's operations.
- Identity: unique identifier (e.g. memory address).

| data type | set of values | example values | operations |
|---|---|---|---|
| `String` | *sequences of characters* | `"Hello, World"`<br>`"I ❤️ COS 126"` | length, concatenate, compare, $i^{\text{th}}$ character, substring,… |
| `Point` | *location in the plane* | (3, 5)<br>(–5, 4) | Euclidean distance, … |

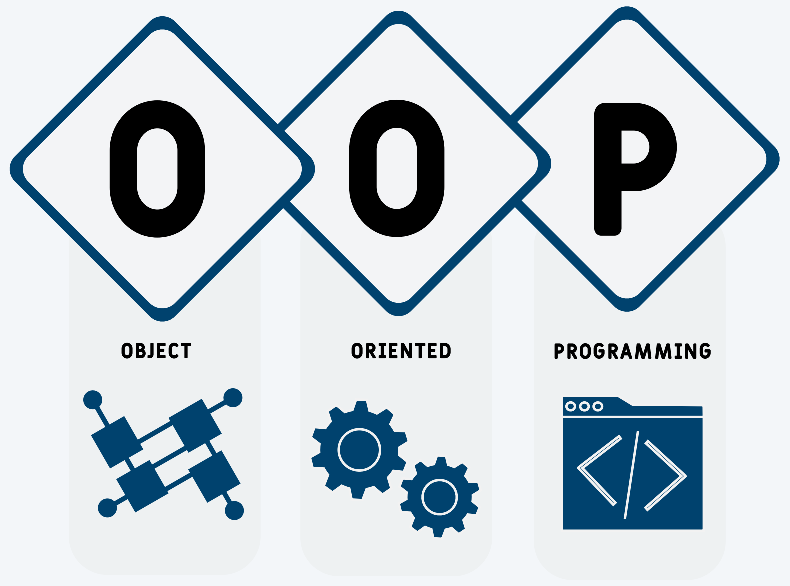
# Object-oriented programming (OOP)

Decomposition. Break up a complex programming problem into smaller functional parts.



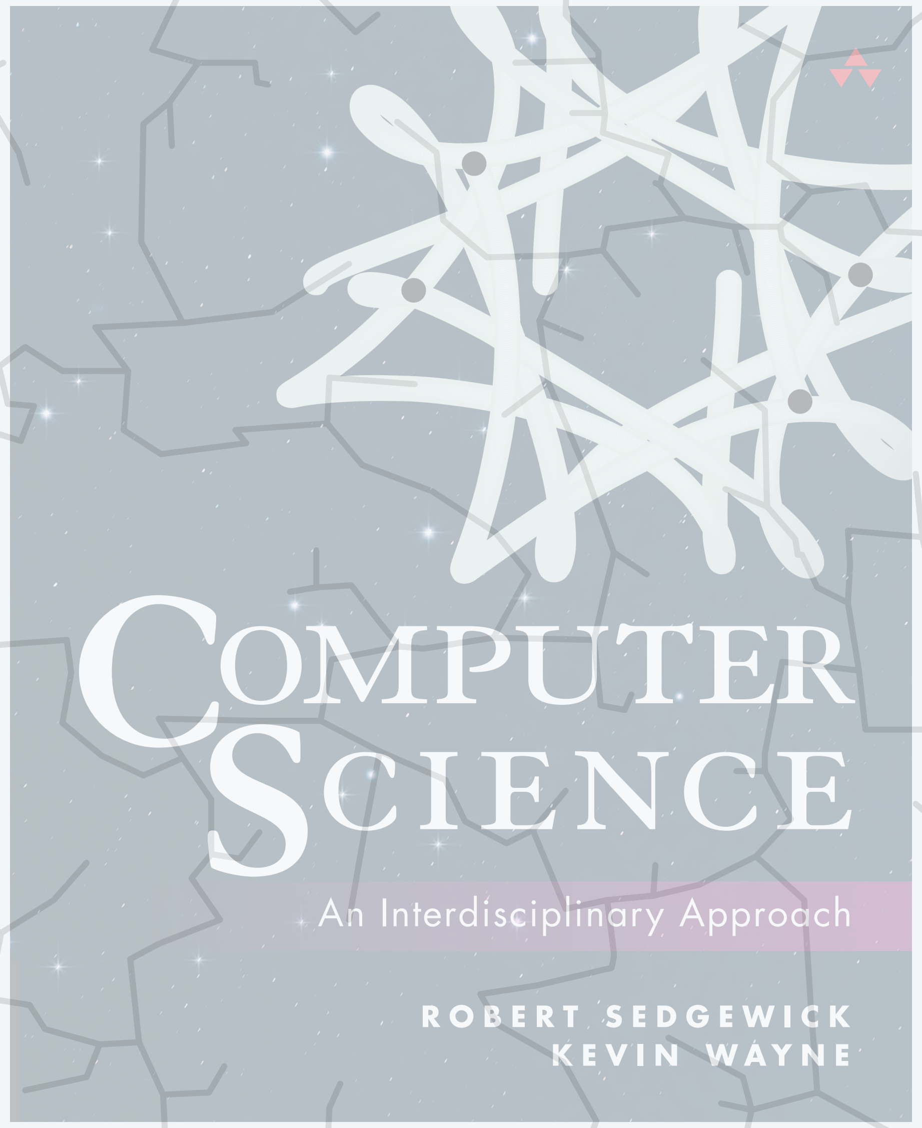
Procedural programming. Implement as a collection of functions.

Object-oriented programming. Implement as a system of interacting objects.




Benefits. Supports the 3 Rs:

- Readability: understand and reason about code.
- Reliability: test, debug, and maintain code.
- Reusability: reuse and share code.




https://introcs.cs.princeton.edu

# 3.3 Designing Data Types

- *encapsulation*
- *immutability*
- *static variables and methods*
- *exceptions*
- *special references*
- *spatial vectors*

# Review: API, client, and implementation

**Application programming interface (API).** Specifies the set of operations for a data type.

**Implementation.** Program that implements a data type's operations.

**Client.** Program that uses a data type through its API.

*contract between client and implementation*

**client**

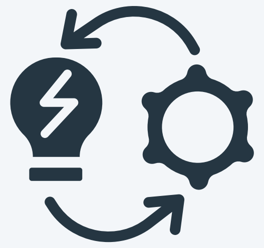
```
Point p = new Point(x0, y0);
...
double dist = p.distanceTo(q);
```

**API**

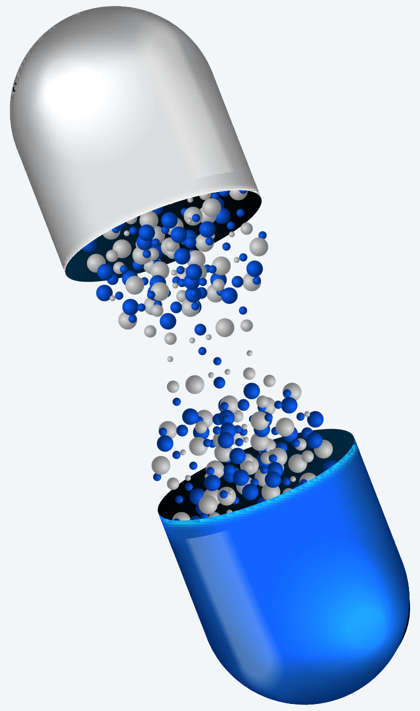
```
public class Point
---------------------------------------------
        Point(double x, double y)
double  distanceTo(Point other)
String  toString()
           ⋮
```

**implementation**

```
public class Point {
   private final double x0;
   private final double y0;
   public Point(double x, double y) {
      x = x0;
      y = y0;
   }
   ...
}
```



# Encapsulation



Encapsulation. Separating clients from implementation details by hiding information.

- Functions encapsulate code.
- Objects encapsulate data and code.

Abstract data type. A data type whose internal representation is hidden from clients.

Principle. A client does not need to know how a data type is implemented in order to use it.

Benefits.

- Can develop client code and implementation code independently.
- Can change implementation details without breaking clients.

*Java 11 changed internal* `String` *representation (to improve performance)*

# The *private* access modifier

## Private access modifier.

- Cannot directly access a *private* instance variable (or method) from another file.
- Compile-time error to attempt to do so.

**implementation**

```
public class Counter {
   private int count;

   public Counter() {
      count = 0;
   }

   public void hit() {
      count++;
   }

}
```

**rogue client**

```
public class RogueClient {
   public static void main(String[] args) {
      Counter counter = new Counter();
      counter.hit();
      ...
      counter.count = -16022;

   }
}
```

*Al Gore received –16,022 votes in Volusia County, Florida in 2000 presidential election*

**compile-time error**

```
~/cos126/oop3> javac-introcs RogueClient.java
RogueClient.java:5:  error: count has
private access in Counter
      counter.count = -16022;
             ^
1 error
```

**Main benefit.** Helps enforce encapsulation. ← *so that programmers (including you!) won't misuse the data type*

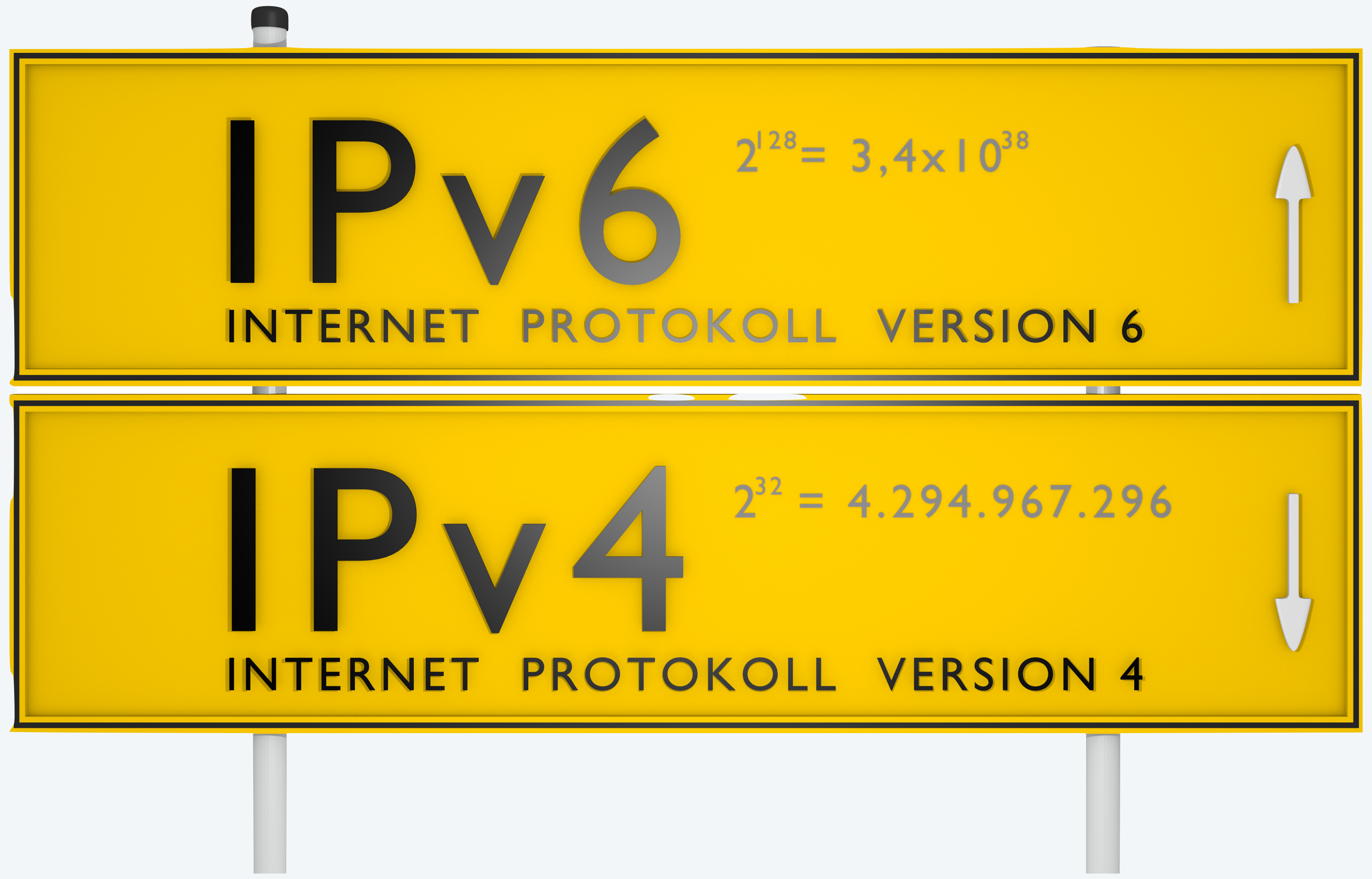
**Best practice.** Declare all instances variables as *private*. ← *requirement in this course*

# Encapsulation fail

## Famous encapsulation failures.

- Y2K bug.
- ZIP code vs. ZIP+4 code.
- IPv4 vs. IPv6.

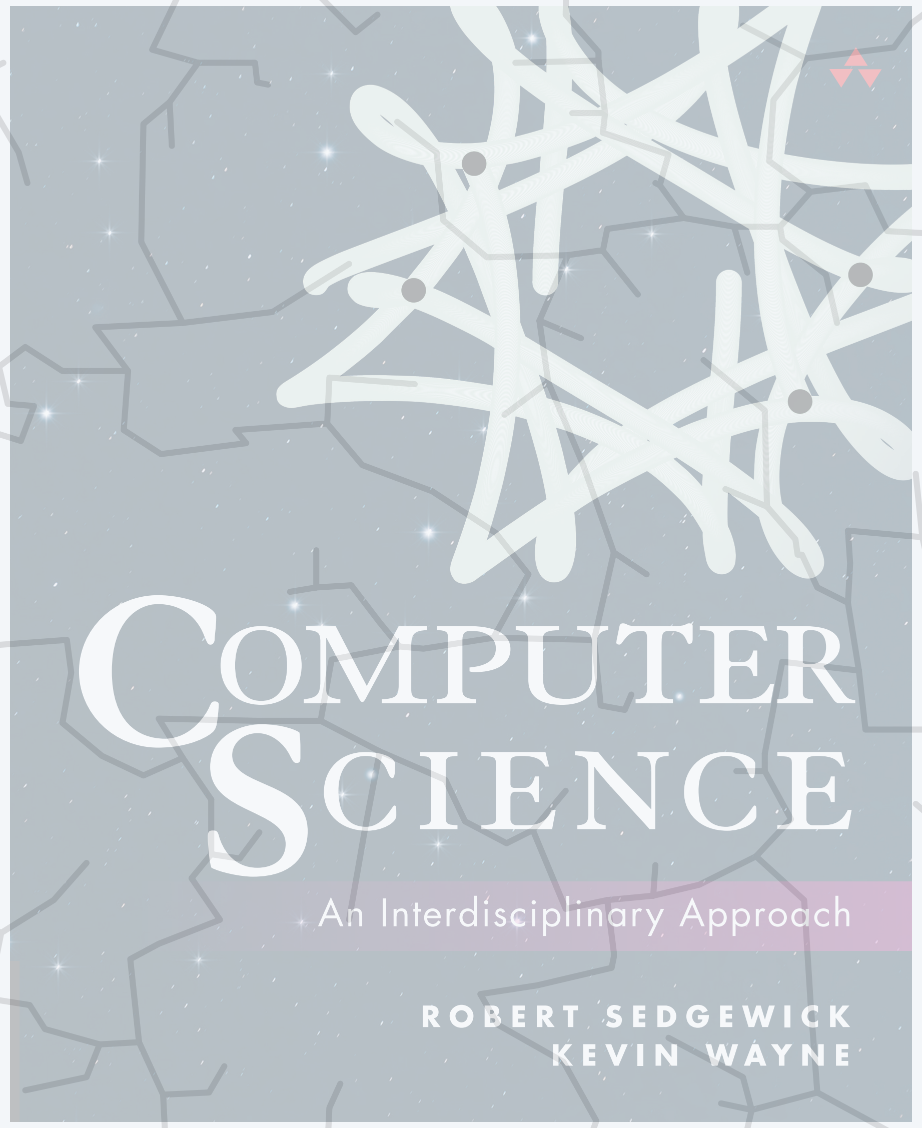




## Designing data types: quiz 1

**Which of the following instance variables should be declared as `private` ?**

- **A.** The instance variables `x` and `y` in `Point`.
- **B.** The instance variables `center` and `radius` in `Circle`.
- **C.** The instance variables `hours` and `minutes` in `Clock`.
- **D.** The instance variables `re` and `im` in `Complex`.
- **E.** All of the above.




https://introcs.cs.princeton.edu

# 3.3 Designing Data Types

- *encapsulation*
- *immutability*
- *static variables and methods*
- *exceptions*
- *special references*
- *spatial vectors*

# Immutability

Immutability. A data type is immutable if you can't change a data-type value once created.

| immutable | mutable |
|---|---|
| `String` | `Clock` |
| `Color` | `Picture` |
| `Point` | `Counter` |
| `Circle` | `int[]` |
| ⋮ | ⋮ |

# Immutability

Immutability. A data type is immutable if you can't change a data-type value once created.

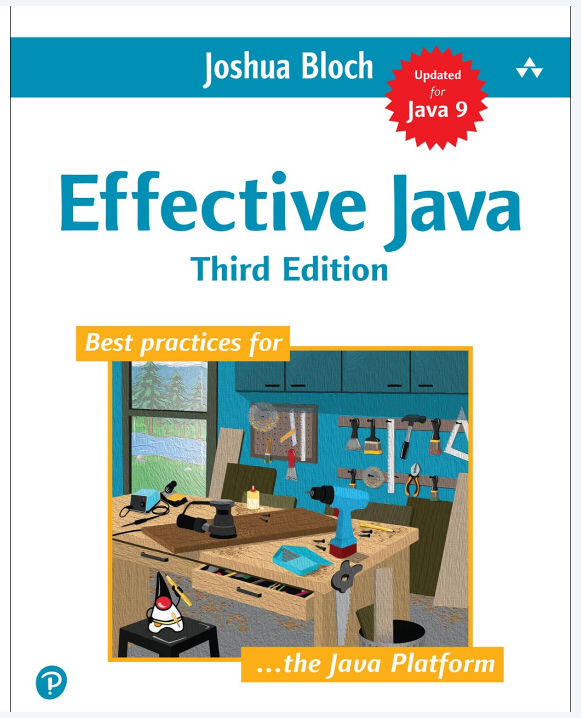
Advantages of immutability.

- Easier to trace, debug, and reason about code.
- Prevents aliasing bugs.
- Simplifies multi-threaded programs.

Main disadvantage. Overhead of creating (and disposing of) extra objects.

Best practices.

> " *Classes should be immutable unless there's a very good reason to make them mutable.... If a class cannot be made immutable, you should still limit its mutability as much as possible.* "
>
> — *Joshua Bloch (Java architect)*



# The *final* access modifier

The access modifier *final* prevents changes to a variable (after initialization).

Ex. Once a point $(x, y)$ is created, cannot change $x$ or $y$.

```
public class Point {
   private final double x;   // x-coordinate
   private final double y;   // y-coordinate

   public Point(double x0, double y0) {
      x = x0;
      y = y0;
   }

   public void scaleX(double alpha) {
      x = alpha * x;
   }

   ...
}
```

*compile-time error*
*(since* x *is* final)

```
~/cos126/oop3> javac-introcs Point.java
Point.java:11: error: cannot assign
a value to final variable x
      x = alpha * x;
      ^
1 error
```

# The *final* access modifier

The access modifier `final` prevents changes to a variable (after initialization).

Advantages.

- Helps enforce immutability.
- Documents that the value will not change.

Best practice. Declare instance variables as `final` (unless compelling reason not to).

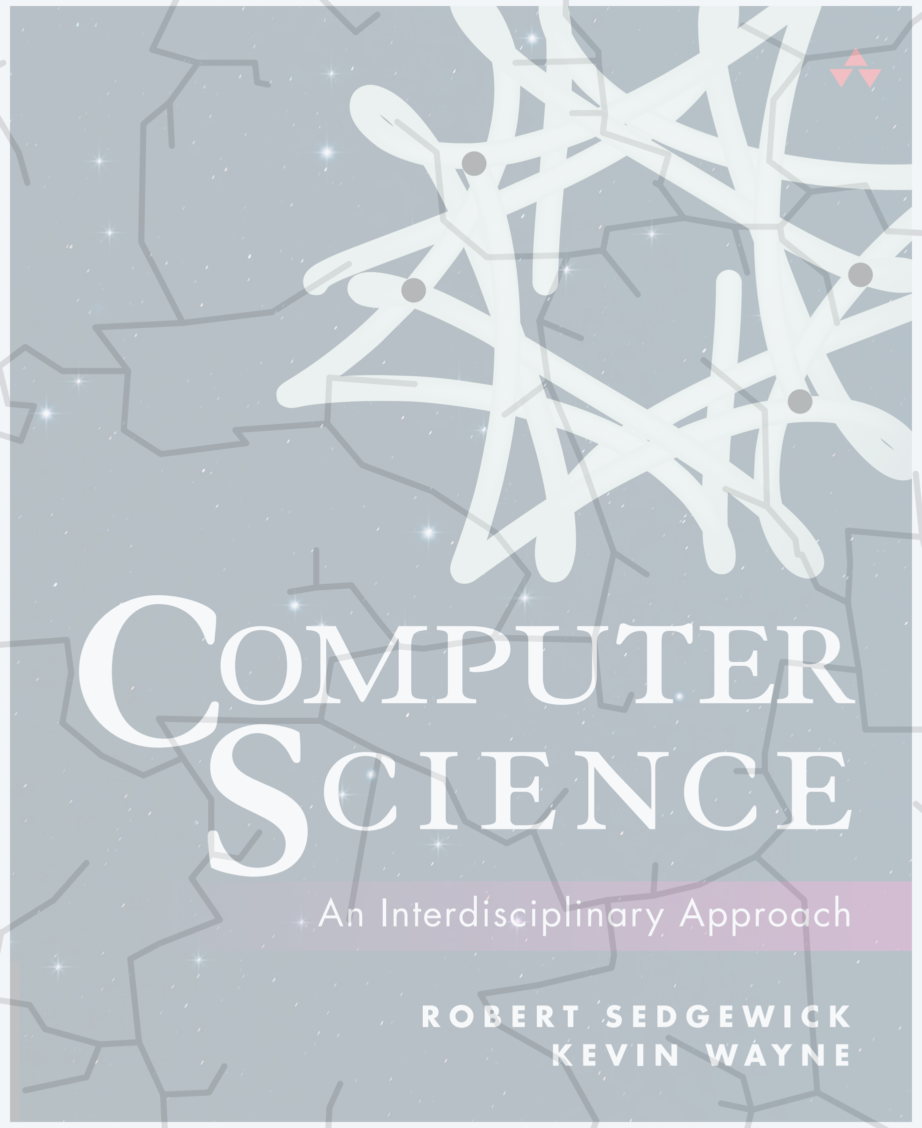
**Which of the following instance variables should not be declared as `final` ?**

**A.** The instance variables `x` and `y` in `Point`.

**B.** The instance variables `center` and `radius` in `Circle`.

**C.** The instance variables `re` and `im` in `Complex`.

**D.** The instance variables `hours` and `minutes` in `Clock`.




https://introcs.cs.princeton.edu

# 3.3 Designing Data Types

- *encapsulation*
- *immutability*
- *static variables and methods*
- *exceptions*
- *special references*
- *spatial vectors*

# Static vs. instance variables

Instance variable. One variable per object.

Static variable. One variable per class.

Common use case. A global constant.

```
public class Clock {

   private static final int MINUTES_PER_HOUR = 60;
   private static final int HOURS_PER_DAY = 24;

   private int hours;     // hours   (0 to 23)
   private int minutes;   // minutes (0 to 59)

   ...

}
```

*one variable per class*

*one variable per object*

Java convention. Define static variables before instance variables.

# Static vs. instance methods

Instance method. Can refer to instance variables / call other instance methods.

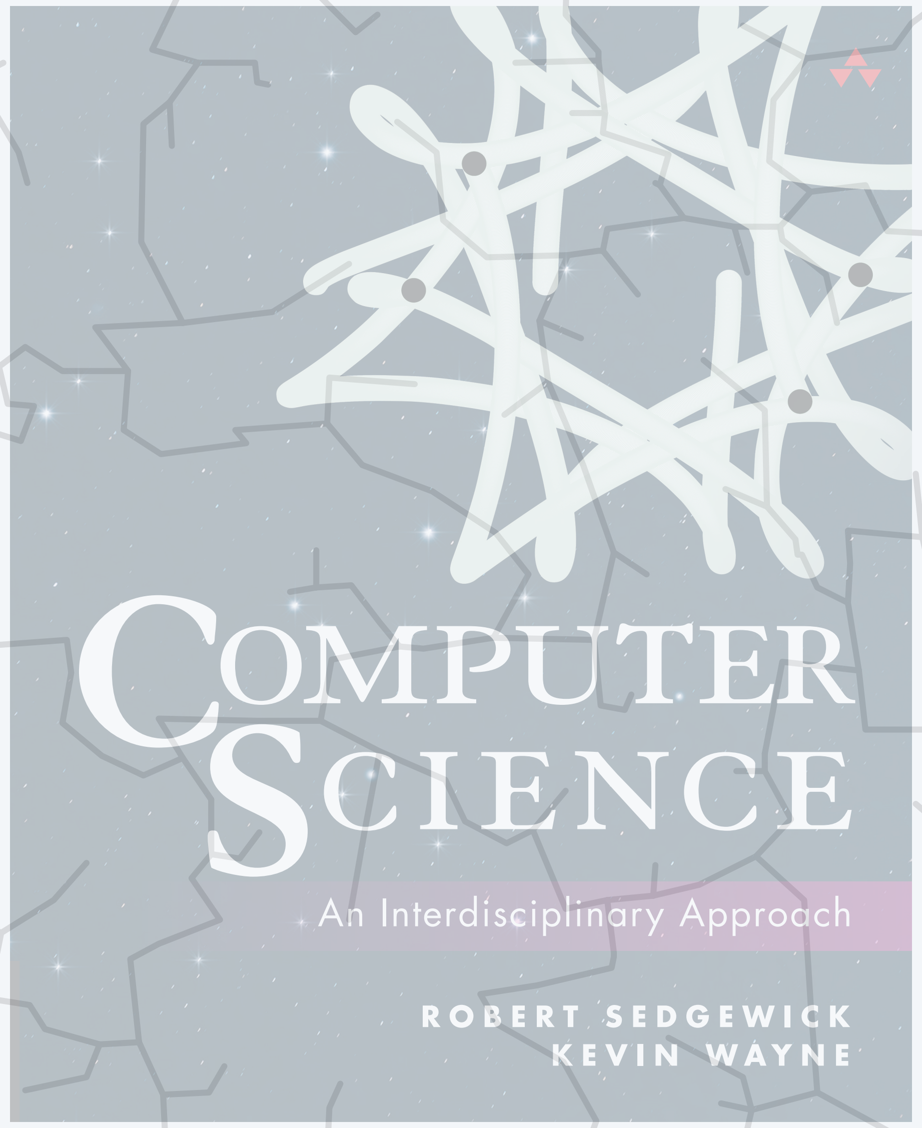
Static method. Cannot refer to instance variables / call instance methods.

```
public class Counter {
   private int count;

   public Counter() {
      count = 0;
   }

   public void hit() {
      count++;
   }

   public static void main(String[] args) {
      hit();
      count++;
   }

}
```

*instance method (associated with an object)*

*static method (associated with the class, not a specific object)*

```
~/cos126/oop3> javac-introcs Counter.java
Counter.java:13: error: non-static method hit()
cannot be referenced from a static context
      hit();
      ^
Counter.java:14: error: non-static variable count
cannot be referenced from a static context
      count++;
      ^
2 errors
```




https://introcs.cs.princeton.edu

# 3.3 DESIGNING DATA TYPES

- *encapsulation*
- *immutability*
- *static variables and methods*
- *exceptions*
- *special references*
- *spatial vectors*

# Exceptions

Exception. A disruptive event that occurs while a program is running, typically to signal an error.

| exception | description | example |
|---|---|---|
| `ArithmeticException` | *performs invalid arithmetic operation* | `1 / 0` |
| `IllegalArgumentException` | *calls constructor/method with invalid argument* | `StdAudio.play("readme.txt")` |
| `NumberFormatException` | *converts string to numeric type* | `Integer.parseInt("12X")` |
| `ArrayIndexOutOfBoundsException` | *accesses array with invalid index* | `a[-4]` |
| `StringIndexOutOfBoundsException` | *accesses string with invalid index* | `s.charAt(s.length())` |
| `NullPointerException` | *uses* `null` *when an object is required* | `null.toString()` |
| ⋮ | ⋮ | |

# Validating arguments

Best practice. If any constructor/method argument is invalid; throw an exception.

```
public Clock(int h, int m) {

   if (h < 0 || h >= HOURS_PER_DAY) {
      throw new IllegalArgumentException("invalid hours");
   }
   if (m < 0 || m >= MINUTES_PER_HOUR) {
      throw new IllegalArgumentException("invalid minutes");
   }

   hours = h;
   minutes = m;
}
```

`throw` *an exception if invalid argument*

```
Clock clock = new Clock(12, -1);
```

**invalid constructor call**

```
~/cos126/oop3> java-introcs BadCallToClock
Exception in thread "main" java.lang.IllegalArgumentException:
invalid minutes
   at Clock.<init>(Clock.java:6)
   at BadCallToClock.main(BadCallToClock.java:4)
```

# Fail-fast principle

Fail-fast principle. Better to abort immediately and noisily (than eventually and silently).

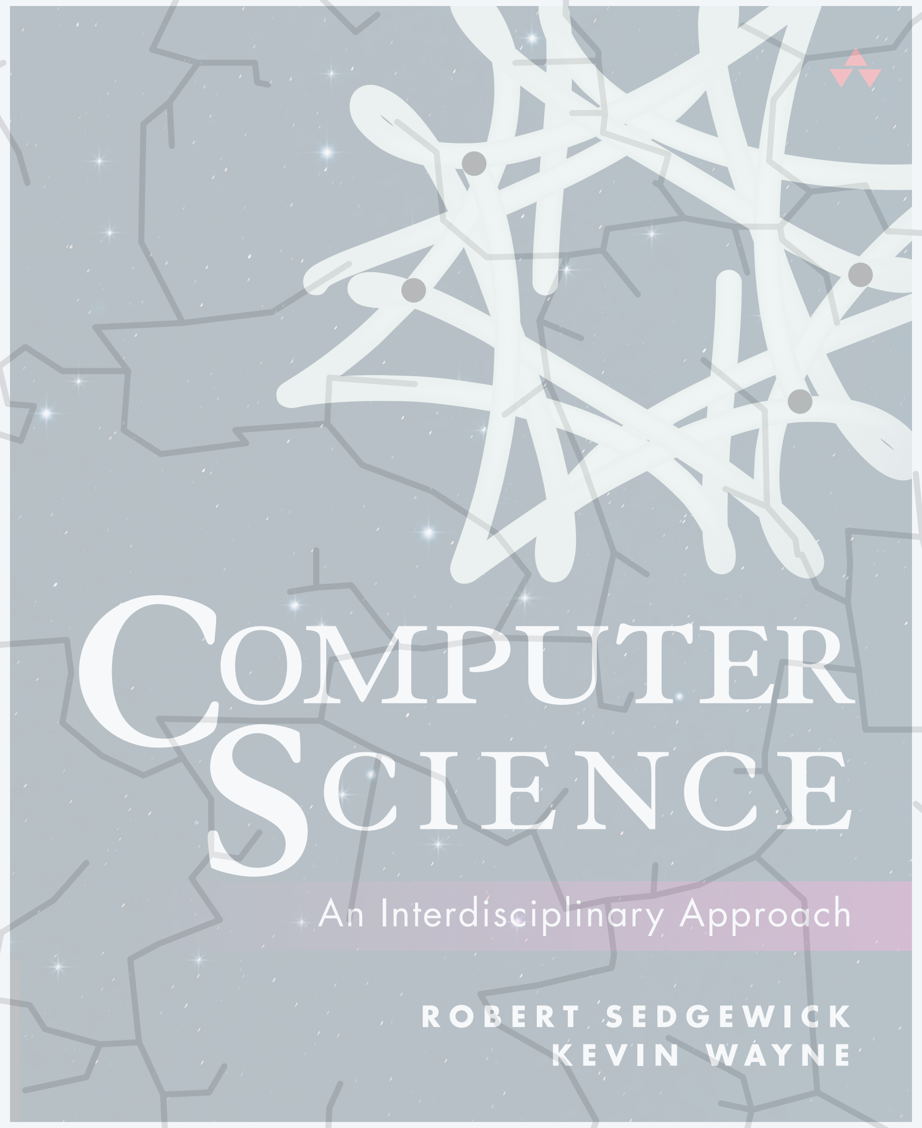

Ex 1. Prefer compile-time error to run-time exception.

Ex 2. Prefer run-time exception to wrong answer.

Cost to fix a bug. Rises steeply over software development cycle.

Silicon Valley meme. “Fail fast, fail often.”

- Experiment freely and learn while trying to achieve objective.
- By quickly finding the failures, you can accelerate learning.




https://introcs.cs.princeton.edu

# 3.3 DESIGNING DATA TYPES

- *encapsulation*
- *immutability*
- *static variables and methods*
- *exceptions*
- *special references*
- *spatial vectors*

# The *null* reference

Null reference. A value that indicates a reference does not refer to any valid object.

- The keyword `null` is a Java literal for the null reference.
- Can assign the value `null` to any variable of a reference type.

```
String s = null;
int len = s.length();
```

*invoke a method or*
*access an instance variable*

Q. What happens if I attempt to manipulate a null reference?

A. Triggers a `NullPointerException`.

Warning. Null references typically arise in practice because instance variables and array elements (of reference types) are auto-initialed to `null`.

# Designing data types: quiz 3

**Which of the following produce a `NullPointerException` ?**

**A.**

```
Mystery x = new Mystery("Hello");
StdOut.println(x.length());
```

**B.**

```
Mystery x = new Mystery("Hello");
StdOut.println(x.distanceToOrigin());
```

**C.** Both A and B.

**D.** Neither A nor B.

```
public class Mystery {
   private Point point;
   private String name;

   private Mystery(String s) {
      String name = s;
   }

   public int length() {
      return name.length();
   }

   public double distanceToOrigin() {
      Point origin = new Point(0.0, 0.0);
      return origin.distanceTo(point);
   }
}
```

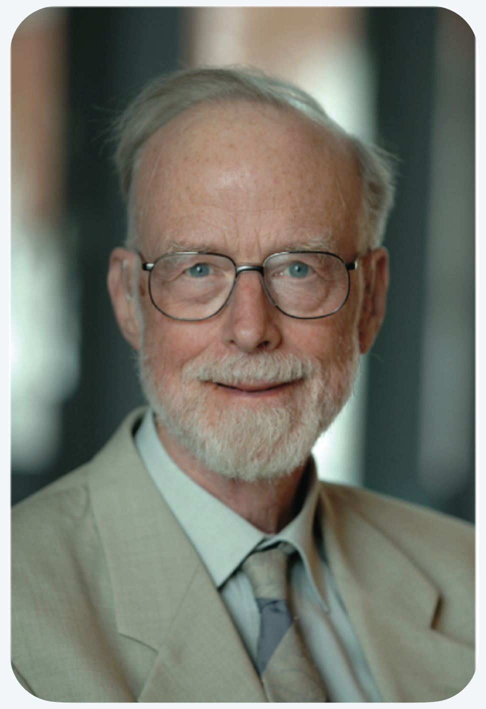
# Tony Hoare quotes

## On null references:

> “ *I call it my billion-dollar mistake. It was the invention of the null reference in 1965… This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.* ”

## On software design:

> “ *There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.* ”



**Tony Hoare**

# The *this* reference

The keyword `this` is a reference to the object whose instance method or constructor is being called.

```java
public class Point {
   private final double x;   // x-coordinate
   private final double y;   // y-coordinate

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

   public double distanceTo(Point that) {
      double dx = that.x - this.x;
      double dy = that.y - this.y;
      return Math.sqrt(dx*dx + dy*dy);
   }

}
```

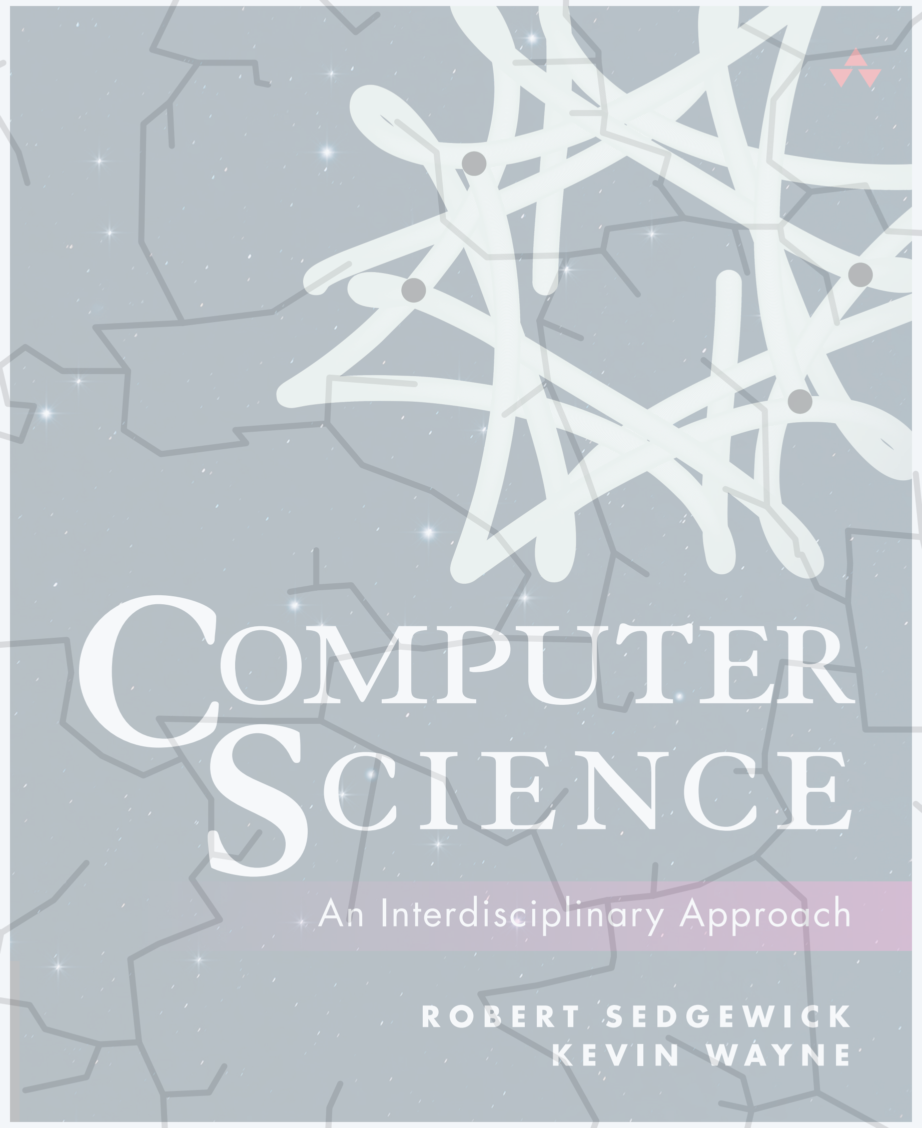
*instance variables of object being constructed*

*"variable shadowing"*

*instance variables of object used to invoke method*

Common use case. Use same names for constructor arguments and instance variables.

Best practice. Programmers debate whether to always (or rarely) use *this*.




https://introcs.cs.princeton.edu

# 3.3 Designing Data Types

- *encapsulation*
- *immutability*
- *static variables and methods*
- *exceptions*
- *special references*
- *spatial vectors*

# Crash course on spatial vectors

A spatial vector is an entity that has magnitude and a direction.

- Quintessential mathematical abstraction.
- Many applications in STEM: force, velocity, momentum, …

Operations on spatial vectors.

- Addition: $\mathbf{x} + \mathbf{y} \;=\; (x_0 + y_0,\; x_1 + y_1,\; \ldots,\; x_{n-1} + y_{n-1})$
- Scaling: $\alpha\,\mathbf{x} \;=\; (\alpha x_0,\; \alpha x_1,\; \ldots,\; \alpha x_{n-1})$
- Dot product: $\mathbf{x} \bullet \mathbf{y} \;=\; (x_0 \cdot y_0 \;+\; x_1 \cdot y_1 \;+\; \ldots \;+\; x_{n-1} \cdot y_{n-1})$
- Magnitude: $\|\mathbf{x}\| = \sqrt{\mathbf{x} \bullet \mathbf{x}}$

| operation | result |
|---|---|
| $(1, 2, 3) \;+\; (4, 5, 6)$ | $(5, 7, 9)$ |
| $2\,(1, 2, 3)$ | $(2, 4, 6)$ |
| $(1, 2, 3) \;\bullet\; (4, 5, 6)$ | $32$ |
| $\|\,(1, 2, 3)\,\|$ | $\sqrt{14}$ |

# Vector API

A spatial vector is an entity that has magnitude and a direction.

**values**

| vector |
|---|
| $(1, 2, 3)$ |
| $(0, -1, 0.5, 0, 0.25)$ |

**API**

| `public class Vector` | **description** |
|---|---|
| `Vector(double[] coords)` | *create a new spatial vector* |
| `Vector plus(Vector that)` | *sum of this vector and* `that` |
| `Vector scale(double alpha)` | *scalar product of this vector and* `alpha` |
| `double dot(Vector that)` | *dot product of this vector and* `that` |
| `double magnitude()` | *magnitude of this vector* |
| `String toString()` | *string representation* |
| ⋮ | ⋮ |

# Vector implementation: test client

Best practice. Begin by implementing a simple test client that tests all methods.

instance variables

constructors

instance methods

**test client**

```
public static void main(String[] args) {
   double[] x = { 3.0, 4.0 };
   double[] y = { -2.0, 3.0 };
   Vector a = new Vector(x);
   Vector b = new Vector(y);
   StdOut.println("a     = " + a);
   StdOut.println("b     = " + b);
   StdOut.println("a + b = " + a.plus(b));
   StdOut.println("2a    = " + a.scale(2.0));
   StdOut.println("a • b = " + a.dot(b));
   StdOut.println("|a|   = " + a.magnitude());
}
```

```
~/cos126/oop3> java-introcs Vector
a     = (3.0, 4.0)
b     = (-2.0, 3.0)
a + b = (1.0, 7.0)
2a    = (6.0, 8.0)
a • b = 6.0
|a|   = 5.0
```

*what we expect, once the the implementation is done*

# Vector implementation: instance variables and constructor

Instance variables. Define data-type values.

Internal representation. Sequence of real numbers.

*each vector corresponds to its own sequence of real numbers (needs its own array instance variable)*

```
public class Vector {

    private final int n;
    private final double[] coords;

    ...
```

*convenient instance variable (optional)*

# Designing data types: quiz 4

**How to implement `Vector` constructor?**

**A.**

```
public Vector(double[] a) {
   n = a.length;
   coords = a;
}
```

**B.**

```
public Vector(double[] a) {
   n = a.length;
   double[] coords = a;
}
```

**C.**

```
public Vector(double[] a) {
   n = a.length;
   for (int i = 0; i < a.length; i++)
      coords[i] = a[i];
}
```

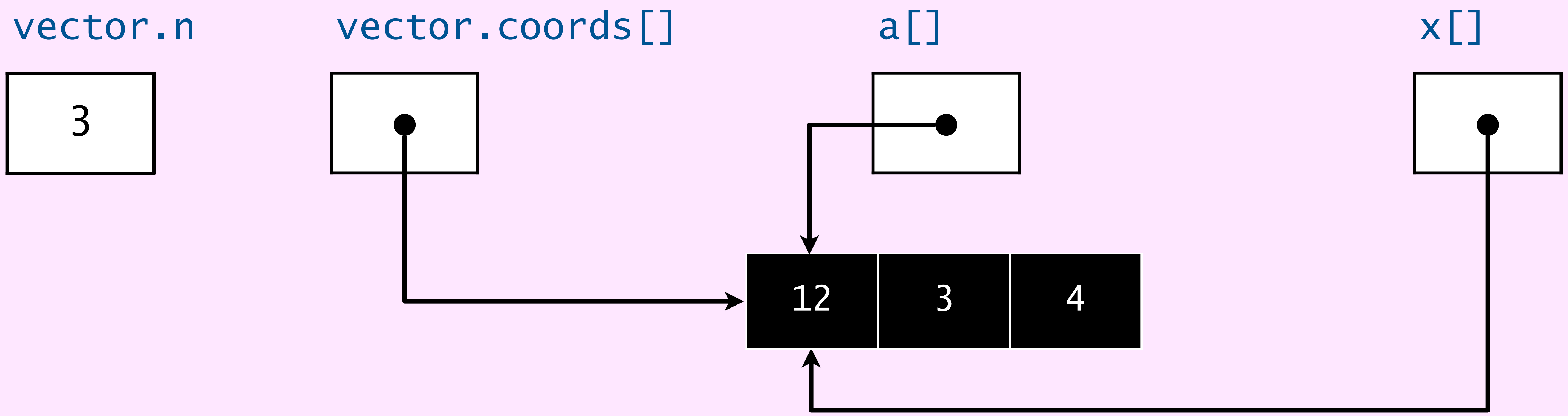
**D.** None of the above.

# Without a defensive copy

```java
public class Vector {
   private final int n;
   private final double[] coords;

   public Vector(double[] a) {
      n = a.length;
      coords = a;
   }
```

```java
double[] x = { 0.0, 3.0, 4.0 };
Vector vector = new Vector(x);
x[0] = -12.0;
StdOut.println(vector.magnitude());
```

$$\sqrt{12^2 + 3^2 + 4^2} = 13$$

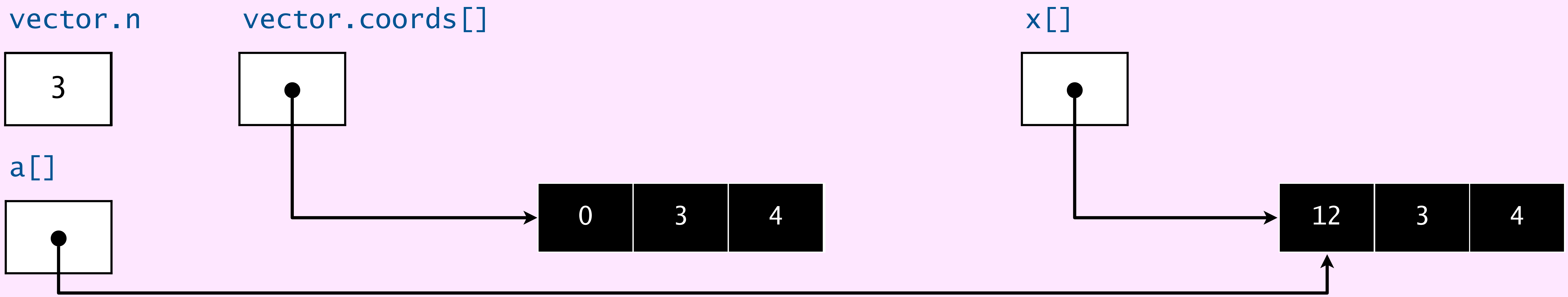


# With a defensive copy

```java
public class Vector {
   private final int n;
   private final double[] coords;

   public Vector(double[] a) {
      n = a.length;
      coords = new double[a.length];
      for (int i = 0; i < a.length; i++)
         coords[i] = a[i];
   }
```

```java
double[] x = { 0.0, 3.0, 4.0 };
Vector vector = new Vector(x);
x[0] = -12.0;
StdOut.println(vector.magnitude());
```

$\sqrt{0^2 + 3^2 + 4^2} = 5$



# Vector implementation:  constructor

Constructors.  Create and initialize new objects.

instance variables

**constructors**

instance methods

test client

```
public class Vector {
   private final double[] coords;
   private final int n;

   public Vector(double[] a) {
      n = a.length;
      coords = new double[n];
      for (int i = 0; i < n; i++) {
         coords[i] = a[i];
      }
   }

   ...
```

*"defensive copy"*

Best practice.  Defensively copy mutable objects.

# Vector implementation: instance methods

Instance methods. Define data-type operations.

```
public class Vector {
   ...

   public Vector plus(Vector that) {
      checkCompatible(this.n, that.n);
      Vector result = new Vector(n);
      for (int i = 0; i < n; i++) {
         result.coords[i] = this.coords[i] + that.coords[i];
      }
      return result;
   }

   private static void checkCompatible(int n1, int n2) {
      if (n1 != n2) {
         throw new IllegalArgumentException("...");
      }
   }

   ...
}
```

*a reusable helper method (can be static)*

instance variables

constructors

**instance methods**

test client

# Vector implementation: instance methods

Instance methods. Define data-type operations.

```
public class Vector {
   ...

   public double dot(Vector that) {
      checkCompatible(this.n, that.n);
      double sum = 0.0;
      for (int i = 0; i < n; i++) {
         sum += this.coords[i] * that.coords[i];
      }
      return sum;
   }

   public double magnitude() {
      return Math.sqrt(this.dot(this));
   }

   ...
}
```

*a rare time where the*
`this` *keyword is indispensable*

instance variables

constructors

**instance methods**

test client

# Vector implementation

*instance variables*

*constructor*

*instance methods*

```
public class Vector {

   private final int n;
   private final double[] coords;

   public Vector(double[] a) {
      n = a.length;
      coords = new double[n];
      for (int i = 0; i < n; i++) {
         coords[i] = a[i];
      }
   }

   public double dot(Vector that) {
      double sum = 0.0;
      for (int i = 0; i < n; i++) {
         sum += this.coords[i] * that.coords[i];
      }
      return sum;
   }

   public double magnitude() {
      return Math.sqrt(this.dot(this));
   }
```

```
   public Vector plus(Vector that) {
      Vector result = new Vector(n);
      for (int i = 0; i < n; i++) {
         result.coords[i] = this.coords[i] + that.coords[i];
      }
      return result;
   }

   public Vector scale(double alpha) {
      Vector result = new Vector(n);
      for (int i = 0; i < n; i++) {
         result.coords[i] = alpha * this.coords[i];
      }
      return c;
   }

   public static void main(String[] args) {
      double[] x = { 3.0, 4.0 };
      double[] y = { -2.0, 3.0 };
      ...
   }

}
```

*test client*

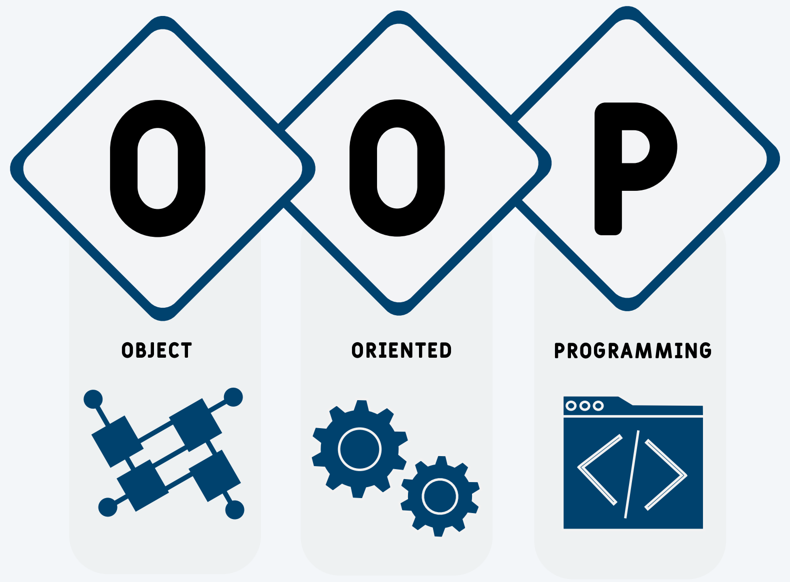
# Summary

Data type. A set of values and a set of operations on those values.

Java class. Java's mechanism for defining a new data type.

Object. An instance of a data type that has

- State: value from its data type.
- Behavior: actions defined by the data type's operations.
- Identity: unique identifier (e.g. memory address).




API, client, implementation. Separate implementation from client via API.

Encapsulation. Hide internal representation of implementation from clients.

Immutability. Data-type values cannot change.

Fail-fast principle. Find errors early in development.

# Credits

| image | source | license |
|---|---|---|
| *OOP* | Adobe Stock | education license |
| *Modular Design* | Modular Management | |
| *Client Avatars* | Adobe Stock | education license |
| *Contract Icon* | Adobe Stock | education license |
| *Implementation Icon* | Adobe Stock | education license |
| *Y2K Bug* | Weekly World News | |
| *ZIP+4 Code* | firstlogic.com | |
| *IP4 vs. IP6* | Adobe Stock | education license |
| *Pharmacy Pill* | Adobe Stock | education license |
| *Private Sign on a Door* | Adobe Stock | education license |
| *Fail Fast* | Adobe Stock | education license |