FUNDAMENTALS

DATA ABSTRACTION

Ruben Acuña

Fall 2018

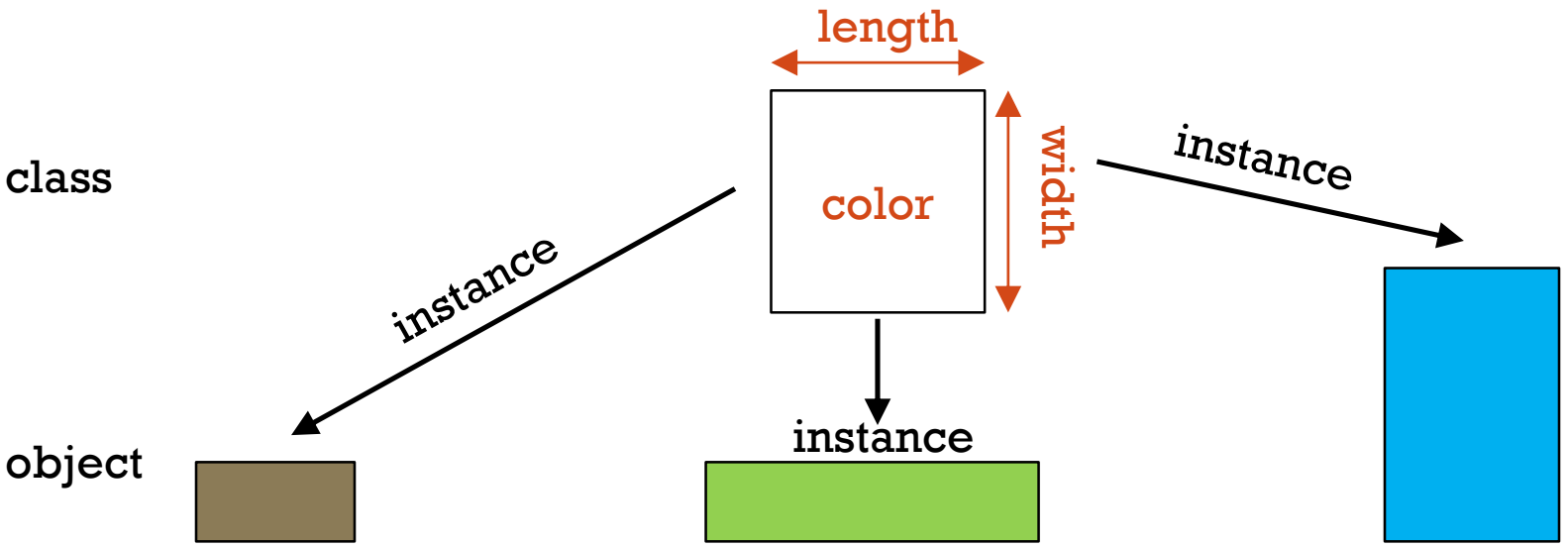
OBJECTS REVIEW

OBJECT-ORIENTED PROGRAMMING

- Classes are a framework (or blueprint) for creating objects that defines the data (but not value), and behavior, they contain.
- Classes are *instanced* to create *objects*.

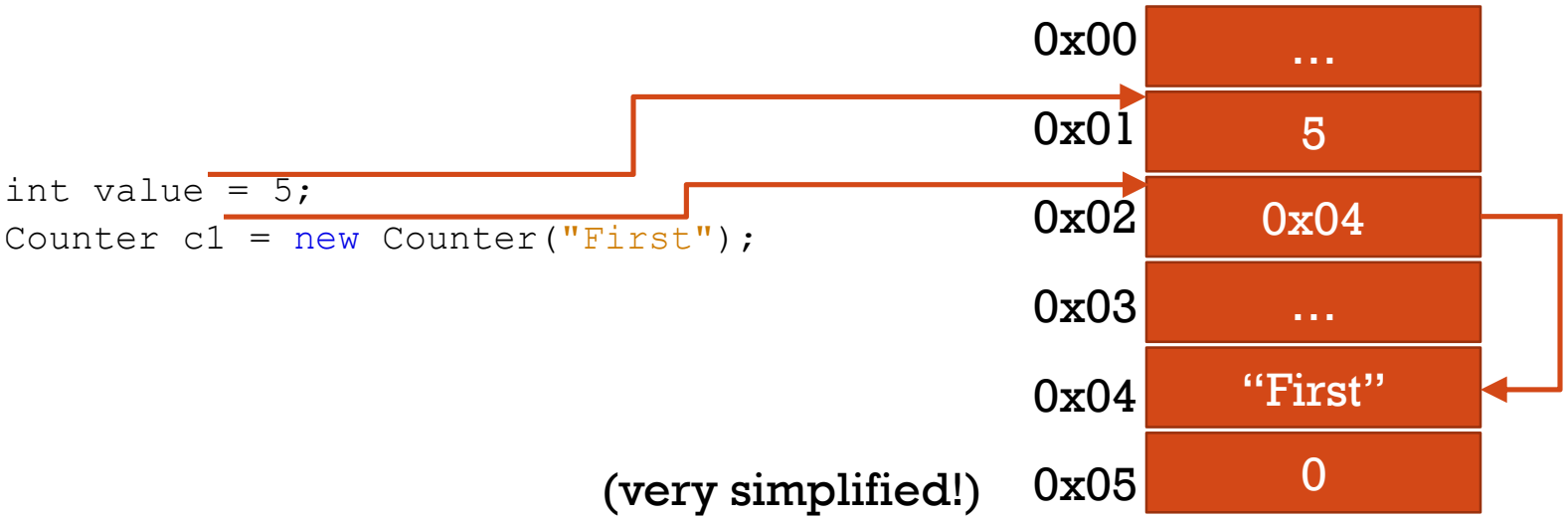
Objects are characterized by:

- State
- Behavior
- Identity



REFERENCE REPRESENTATION

- When a variable like an integer or double is created, it will store the data that represents the number.
- In contrast, object variables do not contain the data representing the object. Instead, they contain the *address* in memory where the object is stored.
- Hence, object values are *references* – they don't contain the object's data, just where to find it.



COUNTER CLASS

```
//from Sedgewick and Wayne
public class Counter {

    private final String name;      // counter name
    private int count = 0;          // current value

    public Counter(String id) {
        name = id;
    }

    public void increment() {
        count++;
    }

    public int tally() {
        return count;
    }

    public String toString() {
        return count + " " + name;
    }
}
```

REFERENCE BEHAVIOR

```
public static void main(String[] args) {
    Counter c1 = new Counter("First");
    Counter c2 = c1;
    c1.increment();
    c2.increment();
    System.out.println(c1);
    System.out.println(c2);
}
```

What will be printed?

```
public static void foobar(Counter c) {
    c.increment();
    System.out.println(c);
}

public static void main(String[] args) {
    Counter c1 = new Counter("First");
    c1.increment();
    foobar(c1);
    System.out.println(c1);
}
```

What will be printed?

"COMMON" METHODS

- All classes in Java are a subclass of *Object*.

- Object includes several methods that can be *overridden* to give a tighter integration with existing functionality.
  - boolean equals(Object obj)
  - int hashCode()
  - String toString()

without toString:

```
System.out.println(point1);  //"Point2@9f239e4a"
```

with toString:

```
System.out.println(point1);  //"(3, 4)"
```

ABSTRACT DATA TYPES

OVERVIEW

- Abstract data type (ADT): a datatype that is represented in terms of operations, and whose internal representation is hidden.
- Naturally defined by objects – but adds emphasis on abstraction.

- A user of an ADT is meant to be isolated from its implementation (e.g., internal presentation). They should care only for the behavior that is exposed (via an API).

- In practice, should we completely ignore the implementation?

A DYNAMIC VIEW

- Rather than rely on a class/object view of ADTs, we can think in terms of operations happening over types (just data):
  - add: Point2D × Point2D → Point2D
  - subtract: Point2D × Point2D → Point2D
  - getX: Point2D → double
  - getDistance: Point2D × Point2D → double
  - equal: Point2D × Point2D → boolean

So what? Well, ADTs have two basic operations:

- Transformations that regenerate the original type (recall closure?).
- Extract information from internal state.

Some ADTs only have the second type.

USAGE

Since ADTs are implemented using classes, they use all the operations allowed for classes. Of interest:

- Creation: `LinkedStack stack = new LinkedStack();`
- Invoke behavior: `stack.push(5);`
- Copy operations (=) work in terms of references as objects.

ADTs vs Classes

- So how is an *ADT* different from a *class*? **Abstraction.**
  - ADT do **not** expose their internal representation (how data is stored and maintained).
  - Classes have no restrictions.
  - Purely a design difference – not enforced with syntax in Java.

ADTs are also different from abstract classes.

ADT EXAMPLES

POINT2D CLIENT

```
public static void main(String[] args) {
    Point2D p1 = new Point2D(3, 5);
    Point2D p2 = new Point2D(3, 5);

    if(p1.equals(p2))
        System.out.println("Dist is 0.");
    else
        System.out.println("Dist is " +
                           p1.distanceTo(p2));

    System.out.println(p1.x());
    System.out.println(p2.theta());
}
```

Point2D API (Sedgewick and Wayne):

```
//Initializes a new point (x, y).
Point2D(double x, double y)

//Returns the x-coordinate.
double x()

//Returns the y-coordinate.
double y()

//Returns the polar radius of this point.
double r()

//Returns the angle of this point in
polar coordinates.
double theta()

//Returns the Euclidean distance between
this point and that point.
double distanceTo(Point2D that)
```

- Does it matter how Point2D is implemented?
- How might Point2D represent data?

POINT2D IMPLEMENTATION

Is *final* without assignment okay?

```
//from Sedgewick and Wayne
public final class Point2D {
    private final double x;
    private final double y;

    public Point2D(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double x() {
        return x;
    }

    public double y() {
        return y;
    }

    public double r() {
        return Math.sqrt(x*x + y*y);
    }

    public double theta() {
        return Math.atan2(y, x);
    }

    public double distanceTo(Point2D that) {
        double dx = this.x - that.x();
        double dy = this.y - that.y();
        return Math.sqrt(dx*dx + dy*dy);
    }

    @Override
    public boolean equals(Object other) {
        if (other == this) return true;
        if (other == null) return false;
        if (other.getClass() != this.getClass())
            return false;
        Point2D p = (Point2D) other;
        return this.x() == p.x() &&
               this.y() == p.y();
    }

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

FURTHER READING

Simple ADTs resemble classes you have already seen.

- **Point2D (immutable)**
- Interval1D
- Interval2D
- *Date (example of a practical ADT)*
- Transaction
- *Accumulator (online; example of a mutable ADT)*
- *Strings (example of an ADT you have used)*

IMPLEMENTING ADTS

- Same as classes.
- Pay attention to documentation – the API is how the ADT will be viewed and used.

IMPLEMENTING ADTS

IMPLEMENTING COMMON OBJECT METHODS

Since all objects are subclasses of the Object class in Java, they inherit its methods:

public boolean equals(Object obj)

- Tests for equality between objects.
  - Need to check for null.
  - Need to check for right class.
  - Then can check member variables.

public string toString()

- String representation. Useful for debugging.

public int hashCode()

- Returns a number representing an objects identity. (To be covered later.)

protected Object clone() throws CloneNot…

- Performs a deep copy of an object.

```
//from Sedgewick and Wayne

@Override

   public boolean equals(Object o) {

      if (o == null) return false;

      if (o.getClass() != this.getClass())

         return false;

      Point2D p = (Point2D) o;

      return this.x() == p.x() &&
             this.y() == p.y();

   }

   @Override

   public String toString() {

      return "(" + x + ", " + y + ")";

   }
```

INTERFACES

An *interface* is a list of methods that an implementing class must define.

Benefits:

- Isolate implementation from functionality.
- Can start using interface before implemented.
- Don't need to expose code.

```
public interface IncrementCounter {
        //Increments the counter by one.
        void increment();
        //Returns the increments since creation.
        int tally();
        //Returns a string representation.
        String toString();
}
```

A class can implement multiple interfaces – unlike class inheritance.

What about *abstract classes*?

IMMUTABILITY

- Since ADTs are implemented as objects, they are stored by Java as references.
- This means they are subject to the referential issues outlined earlier.
- A solution: a datatype is said to be *immutable* when its state cannot change after its creation.
- Generally: make member variables final … and do not provide mutator methods.
- Be careful with objects (arrays!) – they are reference types.

```
//from Sedgewick and Wayne
public final class Point2D {
    private final double x;
    private final double y;

    public Point2D(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double x() {
        return x;
    }
```

EXCEPTIONS

- Don't generate an ADT with bad state!
- Throw an exception to notify the program.

```
//from Sedgewick and Wayne
public Point2D(double x, double y) {
    if (Double.isInfinite(x) || Double.isInfinite(y))
        throw new IllegalArgumentException("Coordinates must be finite");
    if (Double.isNaN(x) || Double.isNaN(y))
        throw new IllegalArgumentException("Coordinates cannot be NaN");
```