

Static Members

CPSC 2150

This Lecture

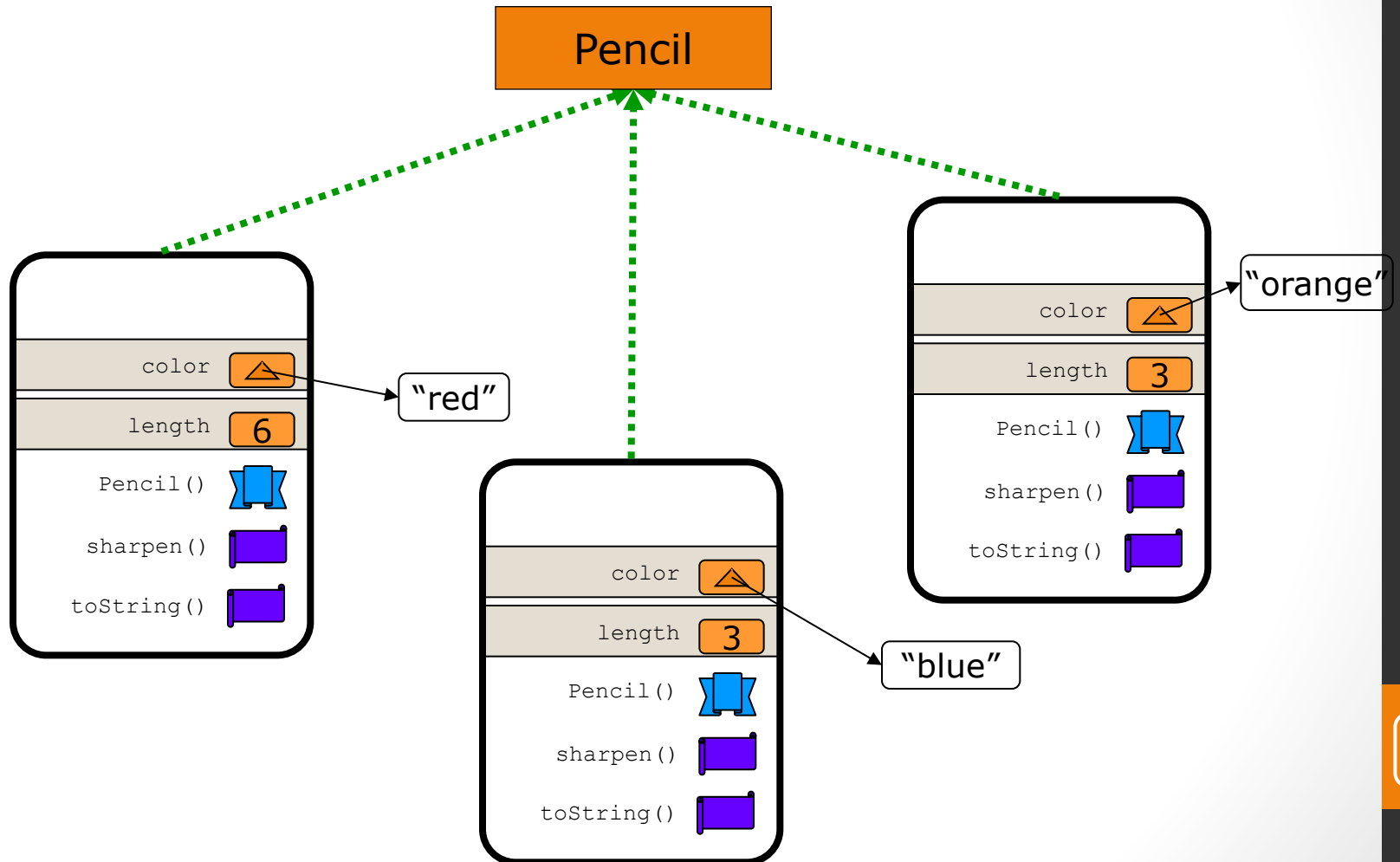
- More details on Java classes and objects
- The focus is on mechanics of doing the following in Java
 - `static` declarations
 - Constants
 - Initialization blocks

Example Class Declaration

- Consider the `Pencil` class. Slightly different (not important).
- Also, invariant is not a focus.

```
public class Pencil {  
    /**  
     *@invariant ...  
     */  
    private String color = "red";  
    private int length;  
  
    public Pencil(int length) {  
        ...  
    }  
}
```

Multiple Pencil Instances



Example Problem

- Suppose we want to count number of `Pencil` instances.

```
public class Pencil {  
    /**  
     *@invariant ...  
     */  
    private String color = "red";  
    private int length;  
  
    public Pencil(int length) {  
        ...  
    }  
}
```

Example Problem

- Here's how we can do it with a `static` declaration.

```
public class Pencil {  
    □    private static int count = 0;  
        private String color = "red";  
        private int length;  
  
        public Pencil (int length) {  
            count++;  
            ...  
        }  
}
```

Object vs Class Members

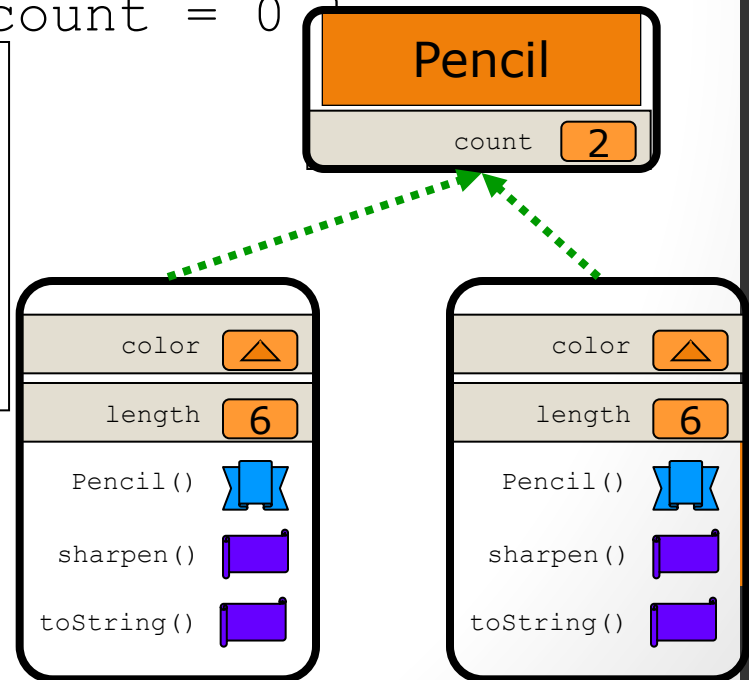
- Class member: only one copy, which is *shared* by all instances

- Keyword: *static*

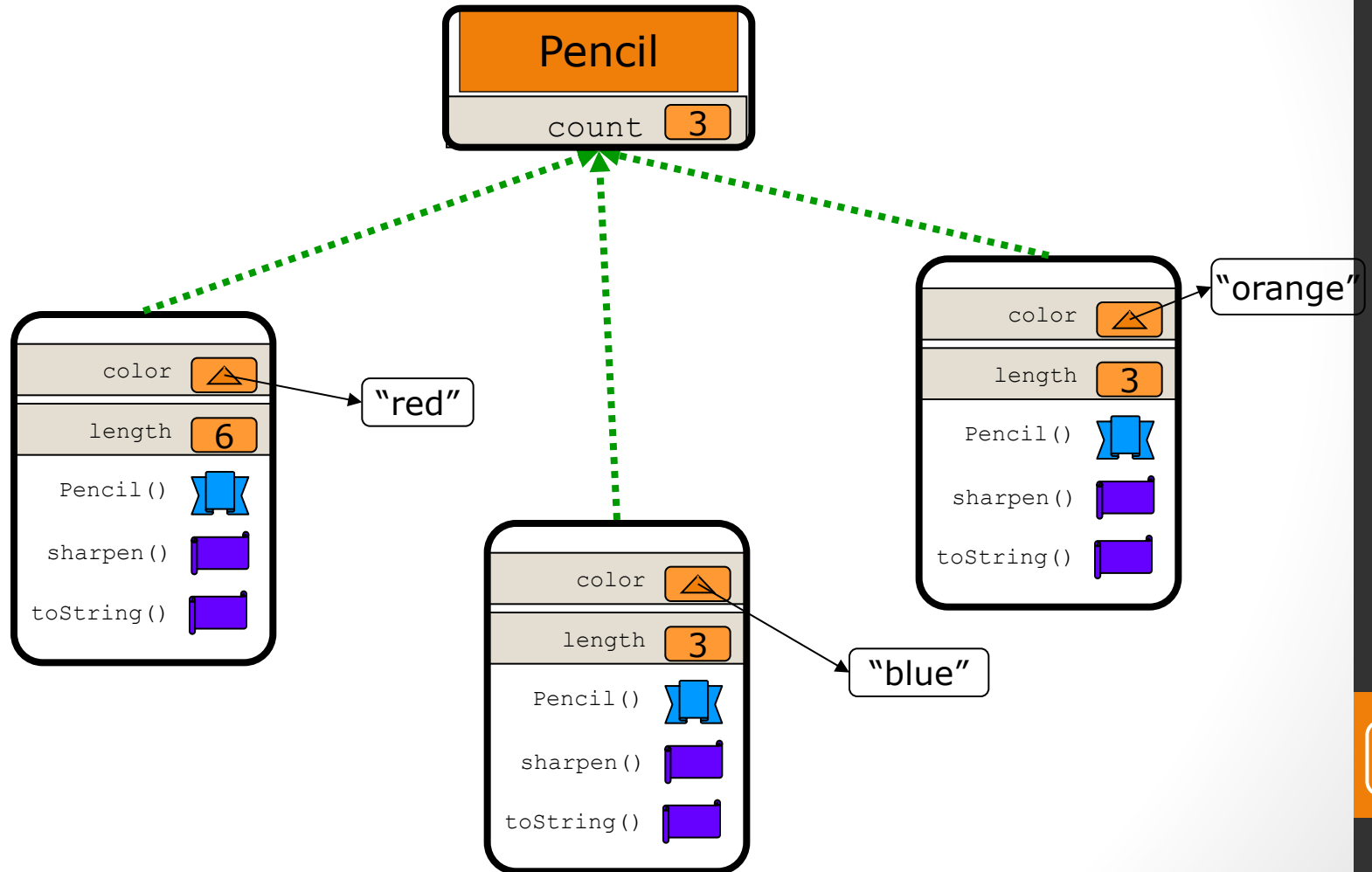
```
static int count;
```

```
static void reset() { count = 0; }
```

```
class Pencil {  
    private static int count = 0;  
    private String color;  
    private int length;  
    . . .  
}
```



Multiple Pencil Instances



aka Instance vs Static Members

- Static members available even before instances (objects) are created!
 - From outside of class: *classname.member*
`Pencil.count++; //must be public`
 - From inside class: *classname* is optional
- Conversely, static members can *not* access instance members
 - Cannot use `length` in `reset` method

Good Practice: Static Members

- Do *not* access static members through object references
- Use class names instead
 - Do this: `int t = Pencil.count;`
 - Not this: `int t = p1.count;`
- This applies within a class too

```
public class Pencil {  
    private static int count = 0;  
    private int length;  
    public void example() {  
        ... = count;           //correct  
        ... = Pencil.count;    //better  
    }  
}
```

Constant Fields: `final`

- Modifier *final* on field means it cannot change

- For primitive type, effectively a constant

```
final int i1 = 53;  
final int i2 = (int) (Math.random()*20);  
final int i3; //constructor must  
              initialize
```

```
. . .  
i2++;
```

← Compile-time Error

- For objects, only the *reference* is constant

```
final Pencil p = new Pencil("blue");
```

```
. . .  
p = new Pencil();  
p.sharpen(3);
```

← Compile-time Error

OK →

- Often used in conjunction with static

- Class-wide constant value

```
static final int DEFAULT_LENGTH = 10;
```

Good Practice: No Magic Numbers

- “Magic Number”: a numeric constant in code

```
for (int i = 0; i < 365; i++) { ... }
```

- Some literals are acceptable
 - Booleans and references (`true`, `false`, `null`)
 - Integers: `-1`, `0`, `1`, `2`

- The rest should *all* be avoided

```
final int DAYS_PER_YEAR = 365;  
for (int i = 0; i < DAYS_PER_YEAR; i++) { ... }
```

- See Java libraries (API, [constant-values](#)):

```
Integer.MAX_VALUE, Math.PI,  
Float.POSITIVE_INFINITY, Thread.MAX_PRIORITY
```

- **Important benefits:**

- Single point of control over change
- Legibility

static, final, public variables

- We can add data members to a class that are
 - `static` – only one copy that belongs to the class and can be accessed via the class name
 - `final` – the value cannot change
 - **Note:** reference type distinction
 - `public` – Available outside of the class
- We only do this with primitive or immutable data types
 - Reference types really aren't `final`
- These are helpful for providing values that the client of a class may need to know or check against
 - Maximum and minimum allowable values for input validation

public data members!

- This is the exception to the `private` data member rules
- Since they are `final`, we don't have to worry about a client changing a value and breaking the code.
- Why `static`?
 - We only want to expose data that is the same for every instance of the class
 - If it can be different for each instance, then it wouldn't be constant, it would be set at some point
 - **Note:** Java allows you to declare a variable as `final`, then set it later. We don't want to make those `public`.
- We really only do this to replace magic numbers, and to make the information easy to find.

static methods

- Your methods can be declared `static` as well
- That means the method can be called by using the class name, and not declaring an object
- Helpful for utility operations
 - Common operations not tied to any data
- `static` methods only have access to `static` data members, not instance members
 - Can only call methods of the same class if they are also `static`

Example: println

- `System.out.println("Hello");`

- What is *System*?
 - A class from Java library
 - **See API documentation:** `java.lang.System`
- What is *out*?
 - A static field of `System` (available from class)
 - **Type:** reference to an instance of `PrintStream`
- What is *println*?
 - An overloaded method in `PrintStream`
 - Different versions for printing `String`, `int`, `boolean`...

Example: `main()`

```
public class HelloWorldApp {  
    public static void main(String[] args) {  
        . . .  
    }  
}
```

- `public`: so that JVM can run this method
- `static`: no instances of class created (yet)
- `void main(String[])`: required signature
 - JVM looks to invoke the method with this name
- `args`: array of command-line arguments
 - Any name can be used for formal parameter
 - “args” is just Java convention

Initialization Block

- Statement block outside methods/constructors
- Executed *before* the body of any constructor

Without initialization block

```
class Body {  
    private long idNum;  
    private String name = "";  
    private Star orbits;  
    private static long nextID = 0;
```

```
    Body( ) {  
        idNum = nextID++;  
    }
```

```
    Body(String name, Star orbits)  
    {  
        this( );  
        this.name = name;  
        this.orbits = orbits;  
    }
```

```
}
```

With initialization block

```
class Body {  
    private long idNum;  
    private String name = "";  
    private Star orbits;  
    private static long nextID = 0;
```

```
    {  
        idNum = nextID++;  
    }
```

```
    Body(String name, Star orbits)  
    {  
        this.name = name;  
        this.orbits = orbits;  
    }
```

```
}
```

Static Initialization Block

- Similar to initialization block, but:
 - Can only reference static members
 - Executed only once, when class is first loaded

```
class Primes {  
    static int[] primes = new int[4];  
  
    static {  
        primes[0] = 2;  
        for(int i = 1; i < primes.length; i++) {  
            primes[i] = nextPrime(i);  
        }  
    }  
    //declaration of static nextPrime(int) . . .  
}
```

Summary

- Static members (i.e., class members)
 - Instance member belongs to one objects
 - Static member is shared amongst instances
- Initialization blocks, including static initialization