

Classes in Java

CPSC 2150

Object-Oriented Programming

- Fundamental component is an *object*
 - A running program is a collection of objects
- An object encapsulates:
 - State (i.e., data)
 - Behavior (i.e., how state changes)
- Each object is an instance of a *class*
 - Class declaration is a blueprint for objects
 - A class is a component type
 - e.g., `Stack`, `String`, `Person`
 - An object is an instance of that component

```
Pencil mathTool = new Pencil();
```

Good Practice: Files and Classes

- Declare one class per file
- Give file the same name as the class declaration it contains
 - Class `HelloWorldApp` declaration appears in `HelloWorldApp.java`
 - Class `Pencil` is defined in `Pencil.java`

Example Class Declaration

```
public class Pencil {  
  
    private boolean hasEraser;  
    private String color = "red";  
    private int length;  
  
    public int sharpen(int amount) { ... }  
  
    public String getDescription() { ... }  
  
    ...  
}
```

Example Class Declaration

```
public class Pencil {  
  
    private boolean hasEraser;  
    private String color = "red";  
    private int length;  
  
    public int sharpen(int amount) {  
        length = length - amount;  
        hasEraser = (length >= 10);  
  
        return length;  
    }  
}
```

Members

- Two kinds of members in a class declaration

- **Fields**, i.e., data (determine the *state*)

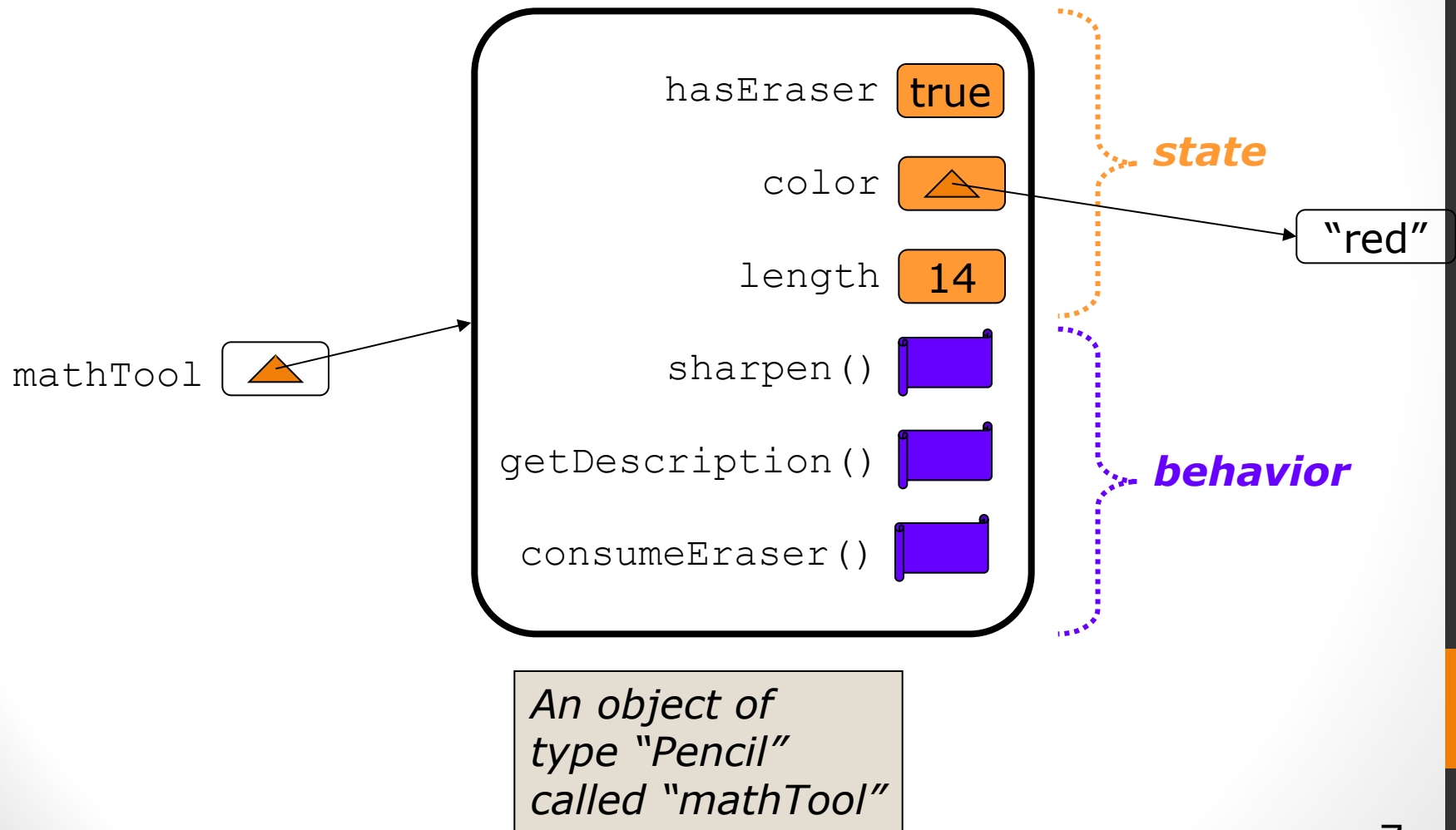
```
private boolean hasEraser;  
private String color;  
private int length;
```

- **Methods**, i.e., procedures (*access/modify* the state)

```
public int sharpen (int amount) {  
    length = length - amount;  
    hasEraser = (length >= 10);  
  
    return length;  
}
```

- **(Much later:** nested classes and nested interfaces)

Graphical View of Object



Object Creation and Deletion

- Explicit object creation with **new**();

```
java.util.Date d = new java.util.Date();  
Integer count = new Integer(34);  
Pencil p1 = new Pencil("red");
```
- Unlike C/C++, memory is *not* explicitly freed
 - References just go out of scope (become null)

```
{  
    // create a Date object (called d)  
    java.util.Date d = new java.util.Date();  
    . . .  
} // d out of scope, object is unreachable
```
 - *Automatic* garbage collection (eventually) deletes unreachable objects
 - There is a call to garbage collection, but it doesn't actually guarantee that garbage collection happens.

Initialization of an Object's Fields

- **Implicit:** Default initial values based on type
 - e.g., `boolean` is *false*, reference type is `null`
`boolean hasEraser; // implicitly false`
- **Explicit:** Initialization with field declaration
`int length = 14;`
- **Special method:** “constructor”
 - **Syntax:** name is same as class, no return type

```
public class Pencil {  
  
    private String color;  
  
    public Pencil(String c) {  
        color = c;  
    }  
  
}
```

 - Invoked by `new ( )`, so can have parameters
 - Runs *after* implicit/explicit field initialization

Default Initial Values

- For fields only
- Does not apply to local variables

<i>Type</i>	<i>Default</i>
<code>boolean</code>	false
<code>byte</code>	0
<code>short</code>	0
<code>int</code>	0
<code>long</code>	0L
<code>float</code>	0.0f
<code>double</code>	0.0d
<code>char</code>	'\u0000'
<code>reference</code>	null

Example Constructor

```
public class Pencil {  
    private boolean hasEraser;  
    private String color;  
    private int length = 14;  
  
    public Pencil(String c) {  
        color = c;  
        hasEraser = (length >= 10);  
    }  
  
    // ... same methods as before ...  
}
```

Visibility

- Members can be `private` or `public`

- member-by-member declaration

```
private String color;
```

```
public int length;
```

```
public int sharpen(int amount) { . . . }
```

- Private members

- Can be accessed only by instances of same class
 - Provide concrete implementation / representation

- Public members

- Can be accessed by any object
 - Provide abstract view (client-side)

Example

```
public class Pencil {  
    private String color;  
    private int length = 14;  
    private boolean isValid(String c) {...}  
    public Pencil(String c, int i) {...}  
    public String toString() {...}  
    public void setColor(String c) {...}  
}
```

```
public class CreatePencil {  
    public void m() {  
        Pencil p = new Pencil("red", 12);  
        p.setColor("blue");  
        p.color = "blue";  
    }  
}
```

OK

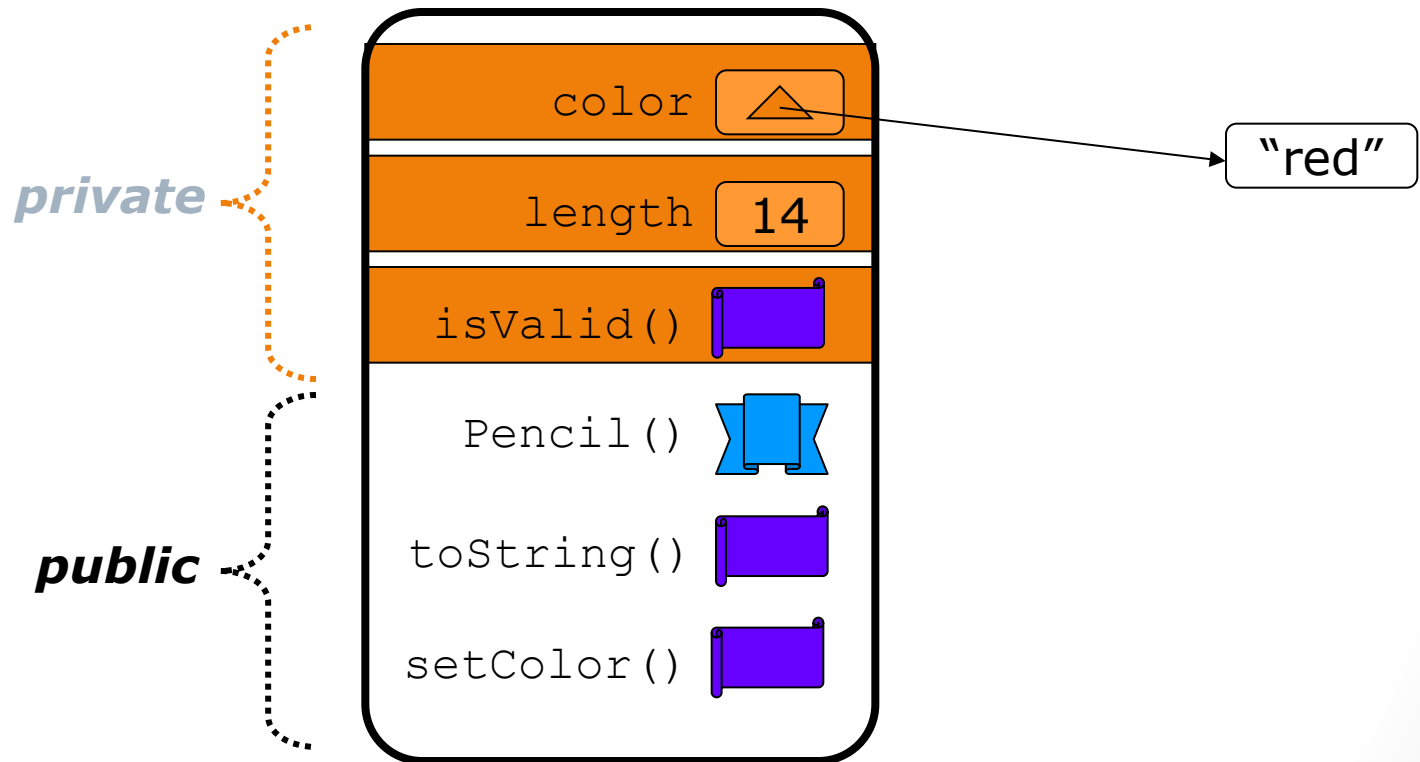


Compile-time Error



(13)

Graphical View of Member Visibility



Good Practice: Member Declarations

- Group member declarations by visibility
 - Java's convention: `private` members at top
- No fields should be `public`
 - **Common (bad) idiom:** `public` “accessor” methods for getting and setting `private` fields (aka getters/setters)

```
public class Pencil {  
    private int length;  
    public int getLength() { . . . }  
    public void setLength() { . . . }  
}
```

- **Better idiom:** Provide `public` members for observing and controlling abstract values of objects
- E.g., `PencilA` and `PencilB` should have *exactly the same accessors* (including signatures)

Method Invocation

- **Syntax:** *objectreference.member*

```
p.color = "red";  
p.toString().length();
```

- Reference is implicit inside same object

```
public class Pencil {  
    private String color;  
    public Pencil() {  
        color = "red";  
    }  
}
```

- Explicit reference to same object available as *this* keyword (from within the object itself)

```
this.color = "red";
```

Method Overloading

- A class can have more than one method with the same name as long as they have different *parameter lists*

```
public class Pencil {  
    . . .  
    public void setPrice(float newPrice) {  
        price = newPrice;  
    }  
    public void setPrice(Pencil p) {  
        price = p.getPrice();  
    }  
}
```

- How does the compiler know which method is being invoked?
 - **Answer:** it compares the number and type of the parameters and uses the matched one

```
p.setPrice(3.4);
```

- Differing *only* in return type is not allowed

Multiple Constructors

- **Default constructor:** no arguments
 - Fields initialized explicitly in declaration or implicitly to language-defined initial values
 - Provided automatically *only* if no constructor defined explicitly

```
public class Pencil {  
    private String color;        // initialized  
                                // implicitly to null  
    private int length = 14;    // initialized  
                                // explicitly  
    ...  
}
```

- **Another constructor:** one same-class argument

```
public Pencil(Pencil p) { . . . }
```
- One constructor can call another with **this()**
 - If another constructor called, must be the first statement

```
public Pencil(Pencil p) {  
    this(p.color); // must be 1st line  
    length = 10;  
}
```

The end

- You are now ready to start creating your own classes in Java
- We'll spend more time in class discussing design decisions and contracts with our classes
- Continue on to the video about the **Object class**