COMP 110/L Lecture 7

Kyle Dewey

Outline

- Introduction to objects
  - Constructors and `new`
  - Instance variables
  - Instance methods
  - `static` vs. non-`static`

Object-Oriented Programming

Basic Idea

The world is composed of *objects*
which interact with each other in well-defined ways

Basic Idea

The world is composed of *objects*
which interact with each other in well-defined ways

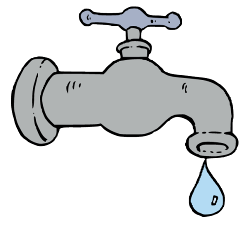
Example: boiling water

–Task: boil water

Basic Idea

The world is composed of *objects*
which interact with each other in well-defined ways

Example: boiling water



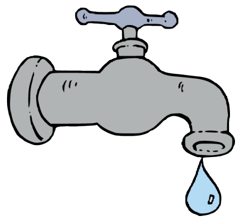
faucet object

-I have a faucet object...

Basic Idea

The world is composed of *objects*
which interact with each other in well-defined ways

Example: boiling water



faucet object

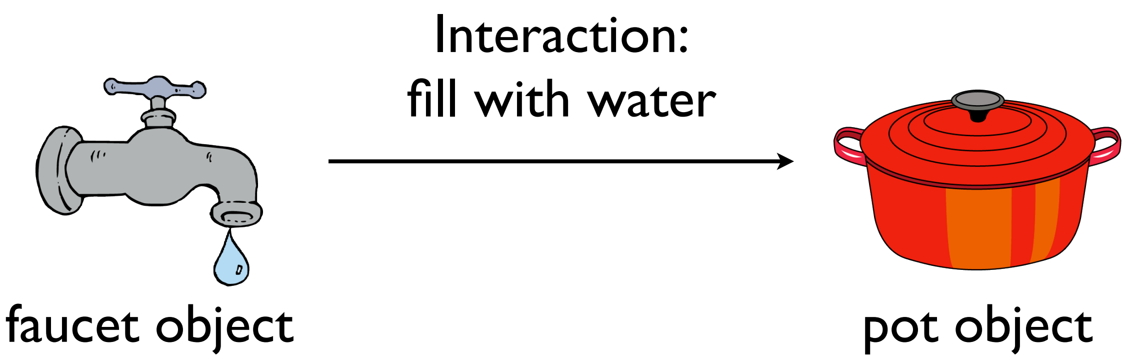
pot object

–...as well as a pot object

Basic Idea

The world is composed of *objects*
which interact with each other in well-defined ways

Example: boiling water



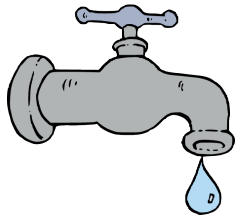
-The faucet can fill the pot

Basic Idea

The world is composed of *objects*
which interact with each other in well-defined ways

Example: boiling water

Interaction:
fill with water



faucet object

$\xrightarrow{\hspace{3cm}}$

pot object

-Now the pot is filled with water

Basic Idea

The world is composed of *objects*
which interact with each other in well-defined ways

Example: boiling water

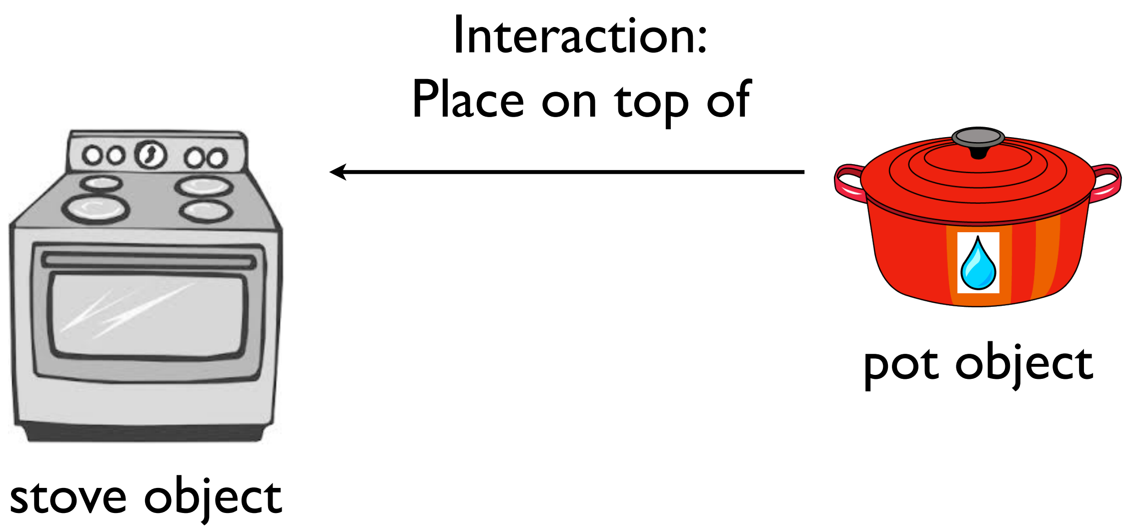
pot object

-Now the pot is filled with water

Basic Idea

The world is composed of *objects*
which interact with each other in well-defined ways

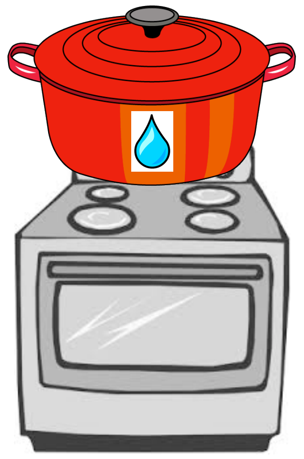
Example: boiling water



Basic Idea

The world is composed of *objects*
which interact with each other in well-defined ways

Example: boiling water



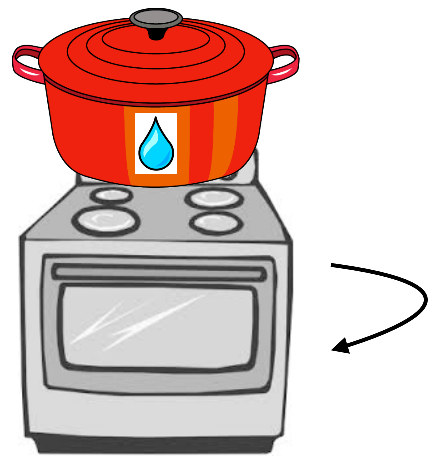
stove object

-The pot is now on top of the stove

Basic Idea

The world is composed of *objects*
which interact with each other in well-defined ways

Example: boiling water



Interaction:
Turn on burner

stove object

-Self-interactions are permitted, and even common

Creating Objects

In Java, we first need a *class* to make an *object*.
A class serves as a blueprint/template for an object.

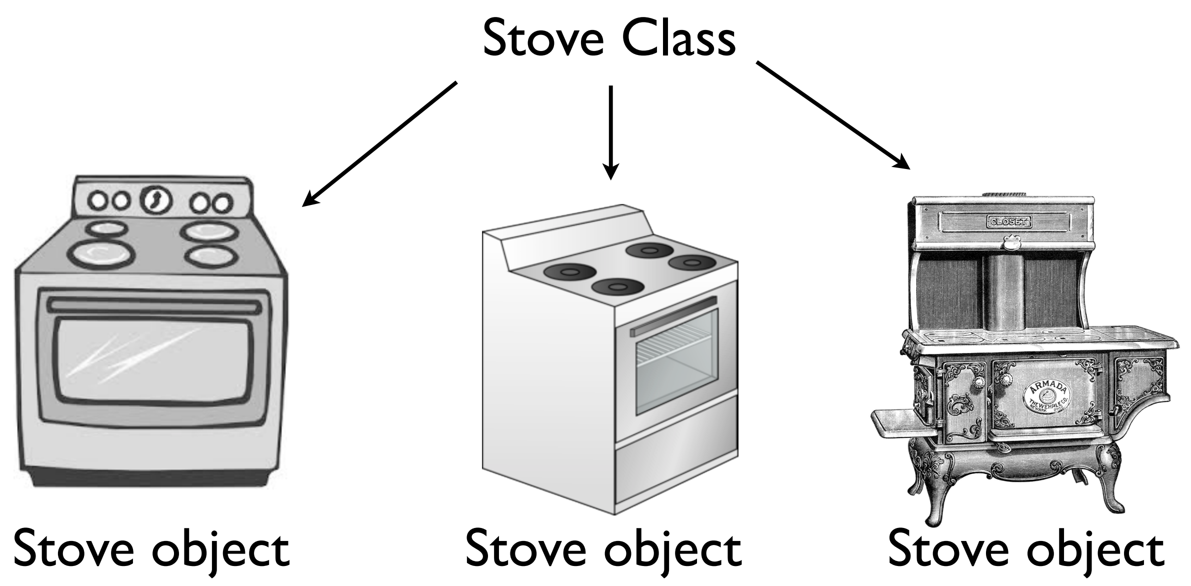
Creating Objects

In Java, we first need a *class* to make an *object*.
A class serves as a blueprint/template for an object.

Stove Class

Creating Objects

In Java, we first need a *class* to make an *object*.
A class serves as a blueprint/template for an object.



-The same class can be used to make different stoves
-These stoves can be different from each other, perhaps even radically different. It all depends on exactly how the class is defined.

`public class`

Declares a `class`, and gives it `public` visibility (more on that later in the course)

-This should sound familiar - you've been using it this whole time!

`public class`

Declares a `class`, and gives it `public` visibility (more on that later in the course)

```
public class Table {
  ...
}
```

Constructors

Constructors

- Code executed upon object creation
- Effectively create the object
- Looks like a method, but no return type (not even `void`) and has the same name as the class

Constructors

- Code executed upon object creation
- Effectively create the object
- Looks like a method, but no return type (not even `void`) and has the same name as the class

```
public class Table {
  public Table() {
    System.out.println(
      "Creating table...");
  }
}
```

Constructors

- Code executed upon object creation
- Effectively create the object
- Looks like a method, but no return type (not even `void`) and has the same name as the class

Constructor

```
public class Table {
  public Table() {
    System.out.println(
      "Creating table...");
  }
}
```

Executing Constructors

`new` executes a given constructor,
creating a new object in the process.

Executing Constructors

`new` executes a given constructor,
creating a new object in the process.

```
Table t = new Table();
```

Example:
`Table.java`

Constructor Parameters

Just like methods, constructors can take parameters

Constructor Parameters

Just like methods, constructors can take parameters

```
public class ConsParam {
  public ConsParam(String str) {
    System.out.println(str);
  }
}
```

Constructor Parameters

Just like methods, constructors can take parameters

```
public class ConsParam {
  public ConsParam(String str) {
    System.out.println(str);
  }
}
```

```
ConsParam p = new ConsParam(“hi”);
```

Example:
`ConsParam.java`

Instance Variables

Instance Variables

Declared in the class.
Each object created from a class (hereafter referred to as an *instance*) has its own instance variables.

Instance Variables

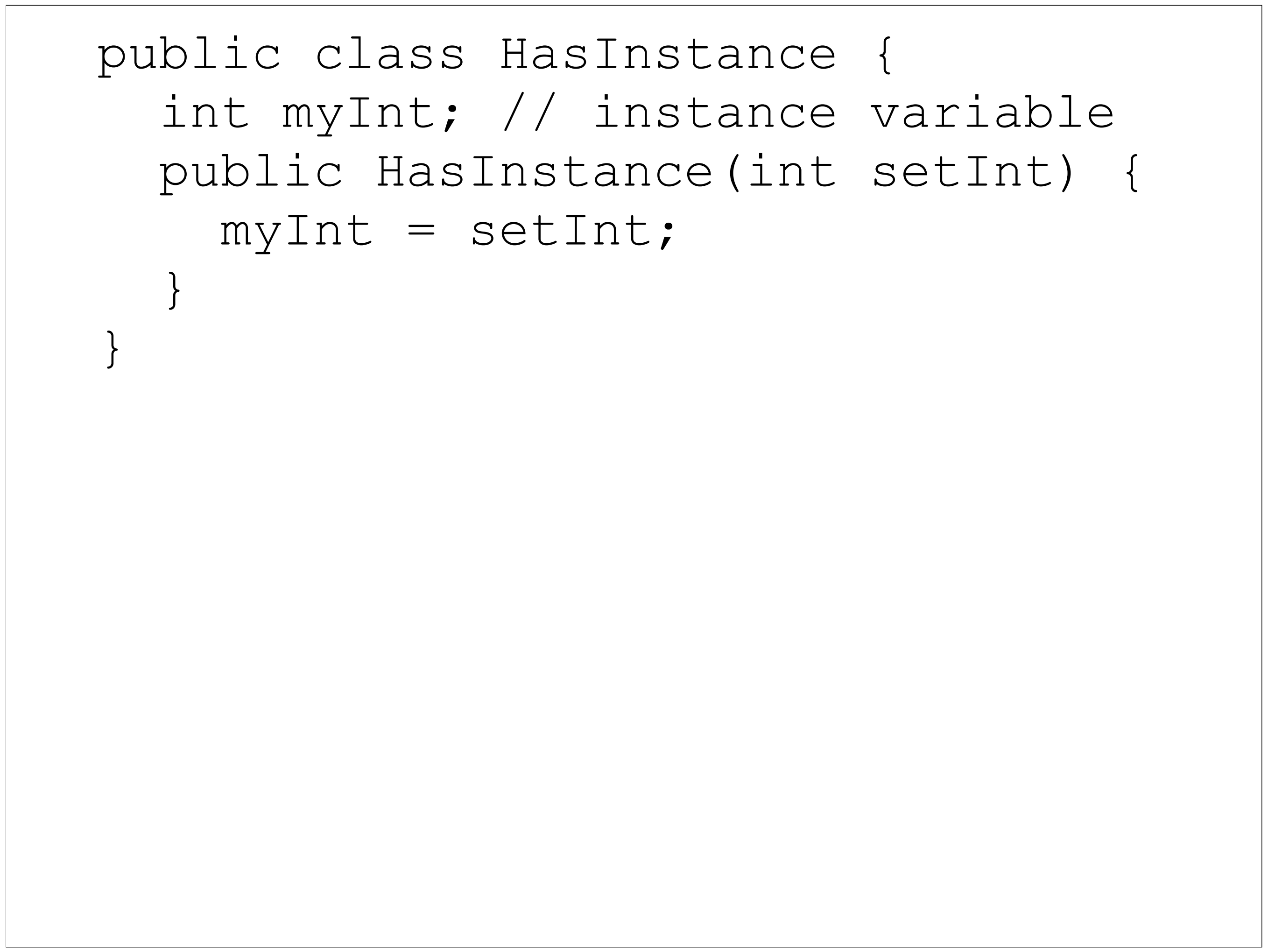
Declared in the class.
Each object created from a class (hereafter referred to as an *instance*) has its own instance variables.

```
public class HasInstance {
  int myInt; // instance variable
  ...
}
```

Instance Variables

Declared in the class.
Each object created from a class (hereafter referred to as an *instance*) has its own instance variables.

```
public class HasInstance {
  int myInt; // instance variable
  public HasInstance(int setInt) {
    myInt = setInt;
  }
}
```



```
public class HasInstance {
  int myInt; // instance variable
  public HasInstance(int setInt) {
    myInt = setInt;
  }
}
```

-Shift up the code to make some room

```
public class HasInstance {
  int myInt; // instance variable
  public HasInstance(int setInt) {
    myInt = setInt;
  }
}
```

```
HasInstance a = new HasInstance(7);
```

-Later on you execute this statement...

```
public class HasInstance {
  int myInt; // instance variable
  public HasInstance(int setInt) {
    myInt = setInt;
  }
}
```

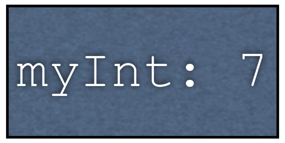
```
HasInstance a = new HasInstance(7);
HasInstance b = new HasInstance(8);
```

-Followed by this statement...

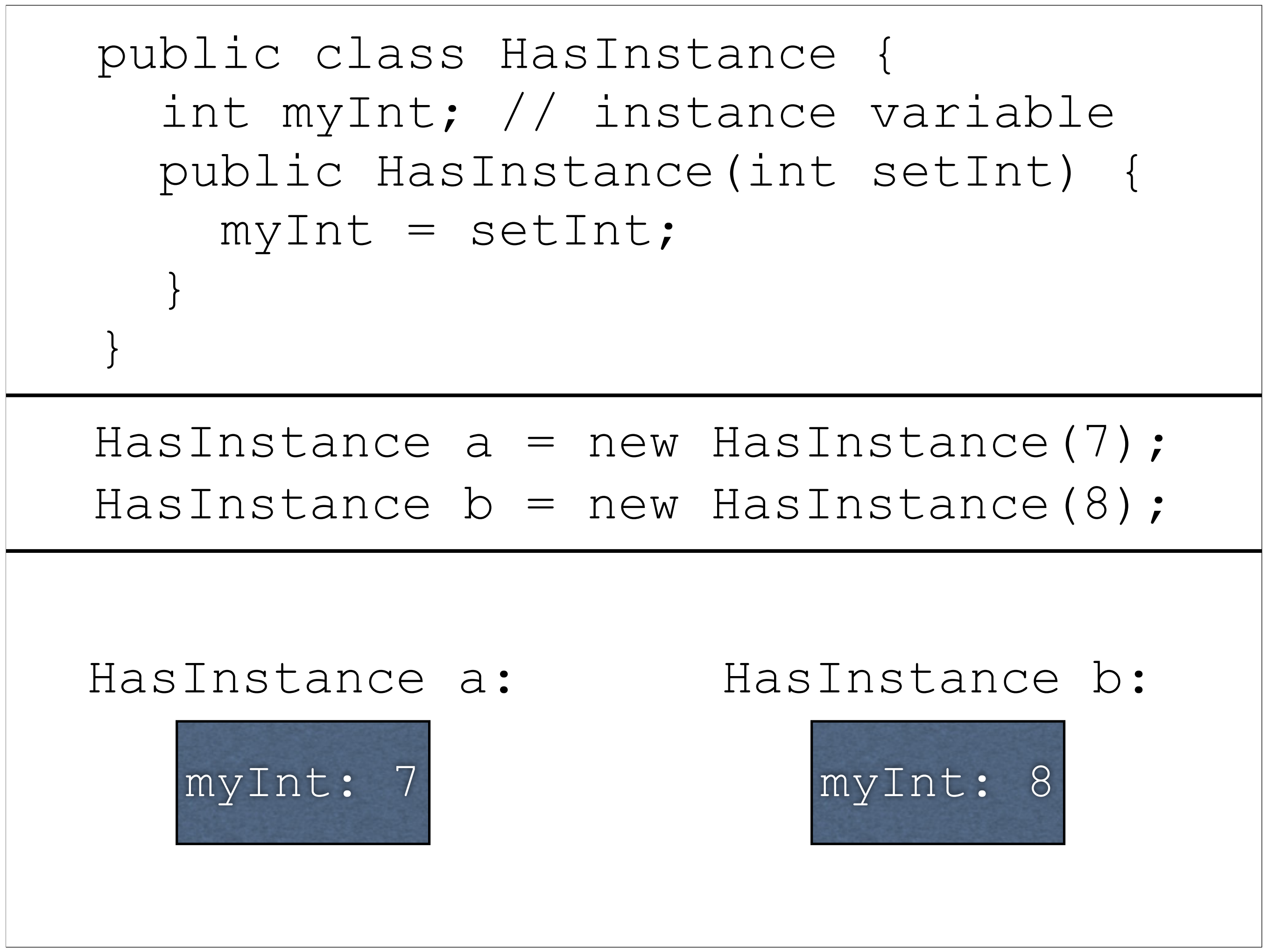
```
public class HasInstance {
  int myInt; // instance variable
  public HasInstance(int setInt) {
    myInt = setInt;
  }
}
```

```
HasInstance a = new HasInstance(7);
HasInstance b = new HasInstance(8);
```

HasInstance a:



-In memory, you'd see that a has its own value of myInt, and that is 7



```
public class HasInstance {
  int myInt; // instance variable
  public HasInstance(int setInt) {
    myInt = setInt;
  }
}
```

```
HasInstance a = new HasInstance(7);
HasInstance b = new HasInstance(8);
```

HasInstance a:

myInt: 7

HasInstance b:

myInt: 8

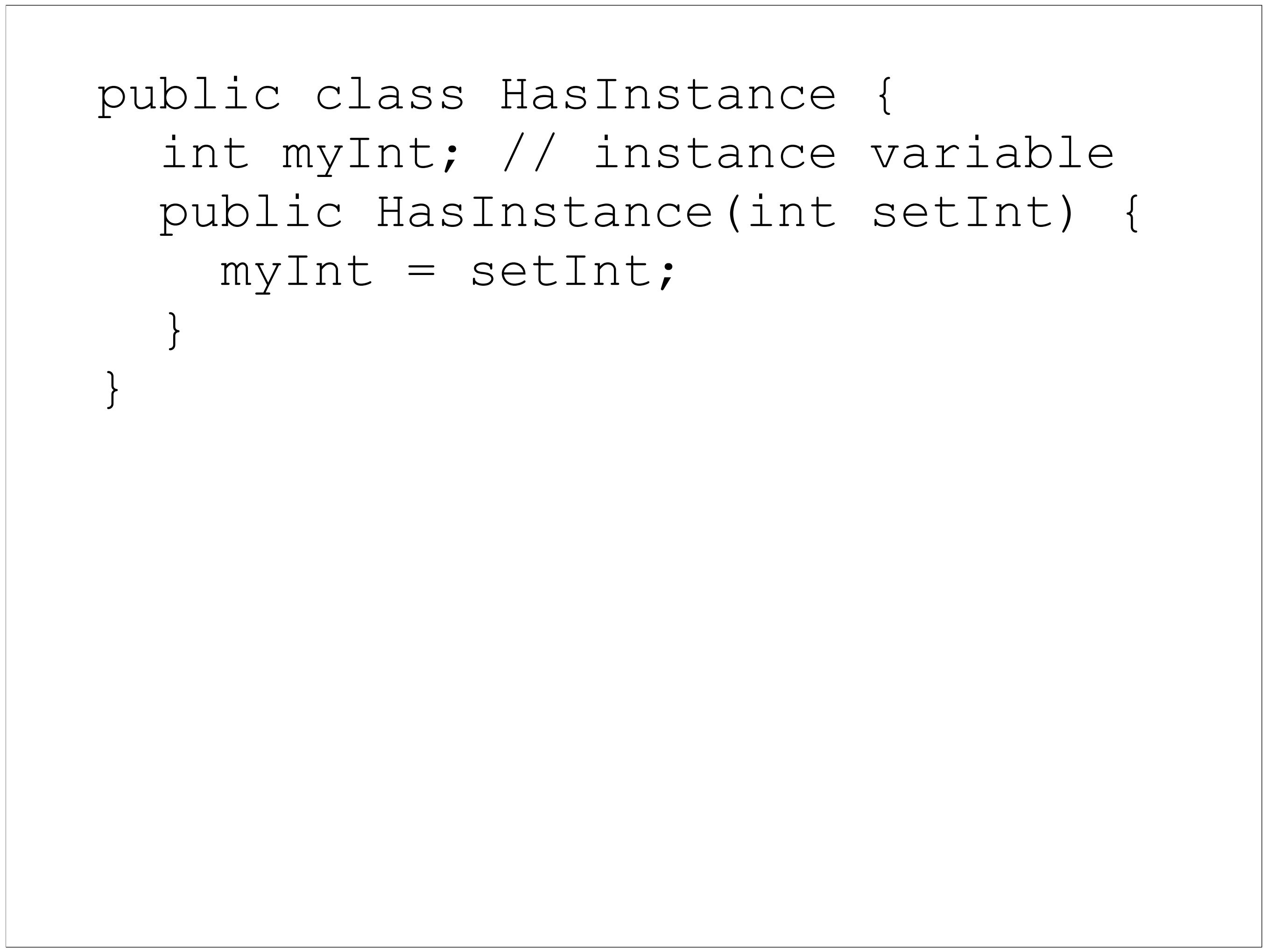
-Similarly, b has its own value of myInt, and that is 8
-Key point: while there is one class, there have been two objects made from this class, and each object has its own values for the instance variable. The instance variables belong to the objects, not the class.

Example:
`HasInstance.java`

Instance Methods

Instance Methods

- Define which interactions can occur between objects
- Declared in the `class`
- Specific to objects created from the class (instances), and operate over instance variables.



```
public class HasInstance {
  int myInt; // instance variable
  public HasInstance(int setInt) {
    myInt = setInt;
  }
}
```

-To show an example, let's take the HasInstance definition from before...

```
public class HasInstance2 {
  int myInt; // instance variable
  public HasInstance2(int setInt) {
    myInt = setInt;
  }

  public void printInt() {
    System.out.println(myInt);
  }
}
```

-...and now we add the printInt instance method
-The name of the class has also been changed, just so we can have both examples in two separate files (namely HasInstance.java and HasInstance2.java)

Example:
`HasInstance2.java`

`static`

Associates something with **the class itself**,
as opposed to individual objects created from the class.

static

Associates something with **the class itself**,
as opposed to individual objects created from the class.

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

–You’ve been defining main and all your methods this way the entire time
–Java forces all source code to be in classes, so this is unavoidable. However, we haven’t really gotten into proper objects yet.

`static` vs. non-`static`

With static: associated with the class.

Without static: associated with objects *created from* the class.

`static` vs. non-`static`

With static: associated with the class.
Without static: associated with objects *created from* the class.

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

`static` vs. non-`static`

With static: associated with the class.
Without static: associated with objects *created from* the class.

With class `MyClass`

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

`static` vs. non-`static`

With static: associated with the class.
Without static: associated with objects *created from* the class.

With class `MyClass`

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

```
public class MyClassTest {
  @Test
  public void someTest() {...}
}
```

`static` vs. non-`static`

With static: associated with the class.
Without static: associated with objects *created from* the class.

With class `MyClass`

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

With objects created from `MyClassTest`

```
public class MyClassTest {
  @Test
  public void someTest() {...}
}
```

Stove Example in Java

- `Water.java`
- `Faucet.java`
- `Pot.java`
- `Stove.java`
- `BoilingWater.java`