Objects and Classes

Recap: Object and Class

- *Classes* are constructs that define objects of the same type.
- An *object*: represents an entity in the real world that can be distinctly identified.

Recap:

- Defining Classes
- The class hierarchy
- Adding variables
- Methods
- Adding Method
- Creating Objects
- Accessing Class member
- Constructor

Constructors

A constructor with no parameters is referred to as a *no-arg constructor*.

- Constructors must have the same name as the class itself.
- Constructors do not have a return type—not even void.
- Constructors are invoked using the new operator when an object is created.

Constructors play the role of initializing objects.

By now you should have a good understanding of what happens in a statement such as:

```
Scanner keys = new Scanner(System.in);
```

We're going to add to our understanding of this statement.

The Scanner() on the right side of the assignment operator is actually a call to a special method called a "**constructor method**".

If you are becoming familiar with using the parenthesis after a method, then this should make sense to you: Scanner() has parentheses, so it does look like it's a method name.

However, for a method, the naming conventions are all wrong!

- The naming conventions for a constructor method are a bit different from other methods.
- A constructor's name must match the class name *exactly* and of course, include a set of parentheses.
- For example, the Book class's constructor methods should be called Book().
- Constructor methods execute whenever a new instance of a class is created.
- The constructor method is responsible for creating the object in memory and, in some cases, giving the instance variables their initial values.

- If you don't include a constructor method in your class, the Java compiler will automatically include the **default constructor** in the compiled bytecode.

- A constructor method, like other methods, can take arguments.

- Usually a programmer would write a constructor with parameter variables so that the class can be instantiated with default values for the instance variables.

Example: a programmer might like to give a new Book a title and price right away instead of saying

```
textBook.title = "Programming Logic";
```

and

```
textbook.price = 59.99;
```

They could do this by using a constructor that takes a string and a double-precision number as arguments:

```
Book logicBook = new Book("Programming Logic", 59.99);
```

For this to work, the Book class must have a constructor method that has a string parameter variable and a double-precision parameter variable.

The constructor can then assign the parameter variable's values to the instance variables. There are some rules that constructor methods must follow:

- The constructor method must always be public.
- The constructor method must *never* have a return type, not even void!
- The constructor method name must always match the class name **exactly**, including capitalization.

So, if we wanted to create the constructor for Book that allowed a programmer to set the title and price as the book object was created, we would start off with a header such as:

```
public Book(String bookTitle, double bookPrice)
 { // rest of code here }
```

Notice that we've followed all the rules - a public method, no return type, and the name matches the name of the class exactly.

Also, we've included the double parameter variables that will contain the values we want to give to the instance variables.

The next step is to set the value of the instance variables to the parameter values:

```
1public Book(String bookTitle, double bookPrice) {
2    title = bookTitle;
3    price = bookPrice;
4}
```

Using the "this" Keyword

What if my constructor method's parameter variables were defined as:

```
public Book(String title, double price) { ... }
```

Would this compile? What about the statements inside the constructor? If we changed them, our constructor would be:

```
1public Book(String title, double price) {
2    title = title;
3    price = price;
4}
```

How does Java know when we're referring to the instance variable title and when we're referring to the parameter variable title?? In fact, the program will compile with this constructor, but it will no longer work properly.

In the statement **`title = title;`**, the variable "title" will be referring to the parameter on both sides of the assignment operator. This means the title (and also price) instance variables won't get their proper values.

Using the "this" Keyword

- The this keyword is the name of a reference that refers to an object itself. One common use of the this keyword is reference a class's *hidden data fields*.
- Another common use of the this keyword to enable a constructor to invoke another constructor of the same class.

This doesn't mean you have to change the parameter variable names back.

There's a way to let Java know that you want to refer to the instance variable and not another variable with the same name. You use the **`this`** keyword.

For example, **`this.title`** refers to "this book object's title", as opposed to the parameter variable title.

So, since we want to put the parameter variable value into the instance variable, it would be correct to say:

```
this.title = title;
```

In other words, "this object's title attribute gets the value in the title parameter." You can update your constructor method so that it now becomes:

```
1public Book(String title, double price) {
2    this.title = title;
3    this.price = price;
4}
```

So is it proper or improper to give your parameters the same names as your instance variables? It's a matter of preference, although it does make the javadocs easier to read (you'll learn to write javadocs in the next course).

Reference the Hidden Data Fields

```
public class F {
  private int i = 5;
  private static double k = 0;

  void setI(int i) {
    this.i = i;
  }

  static void setK(double k) {
    F.k = k;
  }
}
```

```
Suppose that f1 and f2 are two objects of F.
F f1 = new F(); F f2 = new F();

Invoking f1.setI(10) is to execute
   this.i = 10, where this refers f1

Invoking f2.setI(45) is to execute
   this.i = 45, where this refers f2
```

Calling Overloaded Constructor

```
public class Circle {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }

  public double getArea() {
    return this.radius * this.radius * Math.PI;
  }
}
```

this must be explicitly used to reference the data field radius of the object being constructed

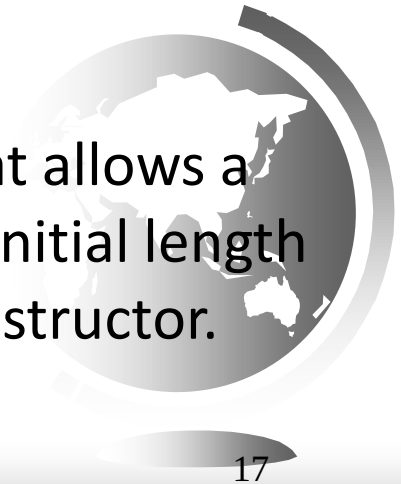
this is used to invoke another constructor

Every instance variable belongs to an instance represented by this, which is normally omitted

Exercise: Design a Room class

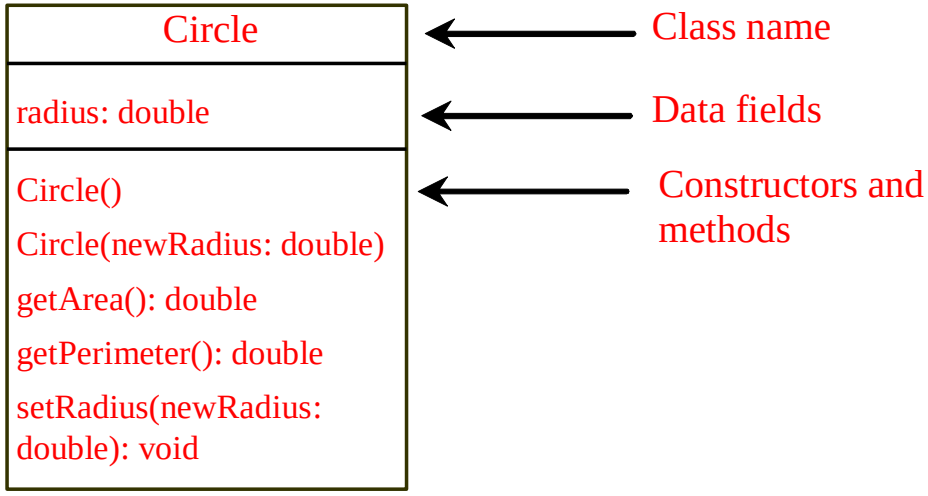
```
public class Room {
  public double length;
  public double width;

  public String getDimensions() {
    return length + " x " + width;
  }

  public double getArea() {
    double area = length * width;
    return area;
  }
}
```

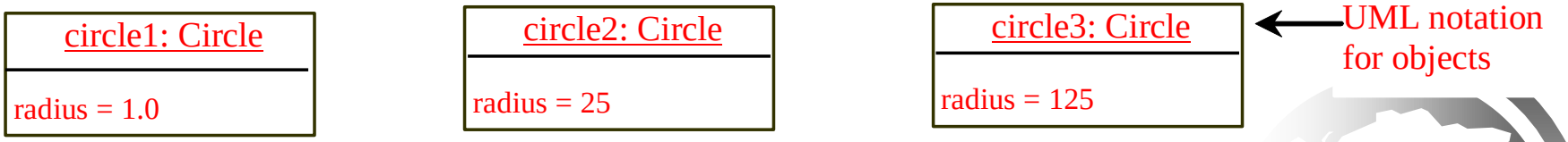
Add a two-parameter constructor to the Room class that allows a programmer to instantiate a new Room object with an initial length and width. Modify your program to try out the new constructor.



UML Class Diagram

UML Class Diagram





Classes

```
class Circle {
  /** The radius of this circle */
  double radius = 1.0;

  /** Construct a circle object */
  Circle() {
  }

  /** Construct a circle object */
  Circle(double newRadius) {
    radius = newRadius;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * 3.14159;
  }
}
```

Data field

Constructors

Method