Classes and Methods Part II

creating objects

invoking methods

String, Random, Math

CREATING OBJECTS

Constructors

- Constructors are used to create and set up objects
  - Remember objects are *instances of* a class.
  - The class is the blueprint, the object is the house.
- Constructors are special methods with
  - **no** return type (not even void)
  - same name as the class
- Constructors are invoked with **`new`**

The **new** Keyword

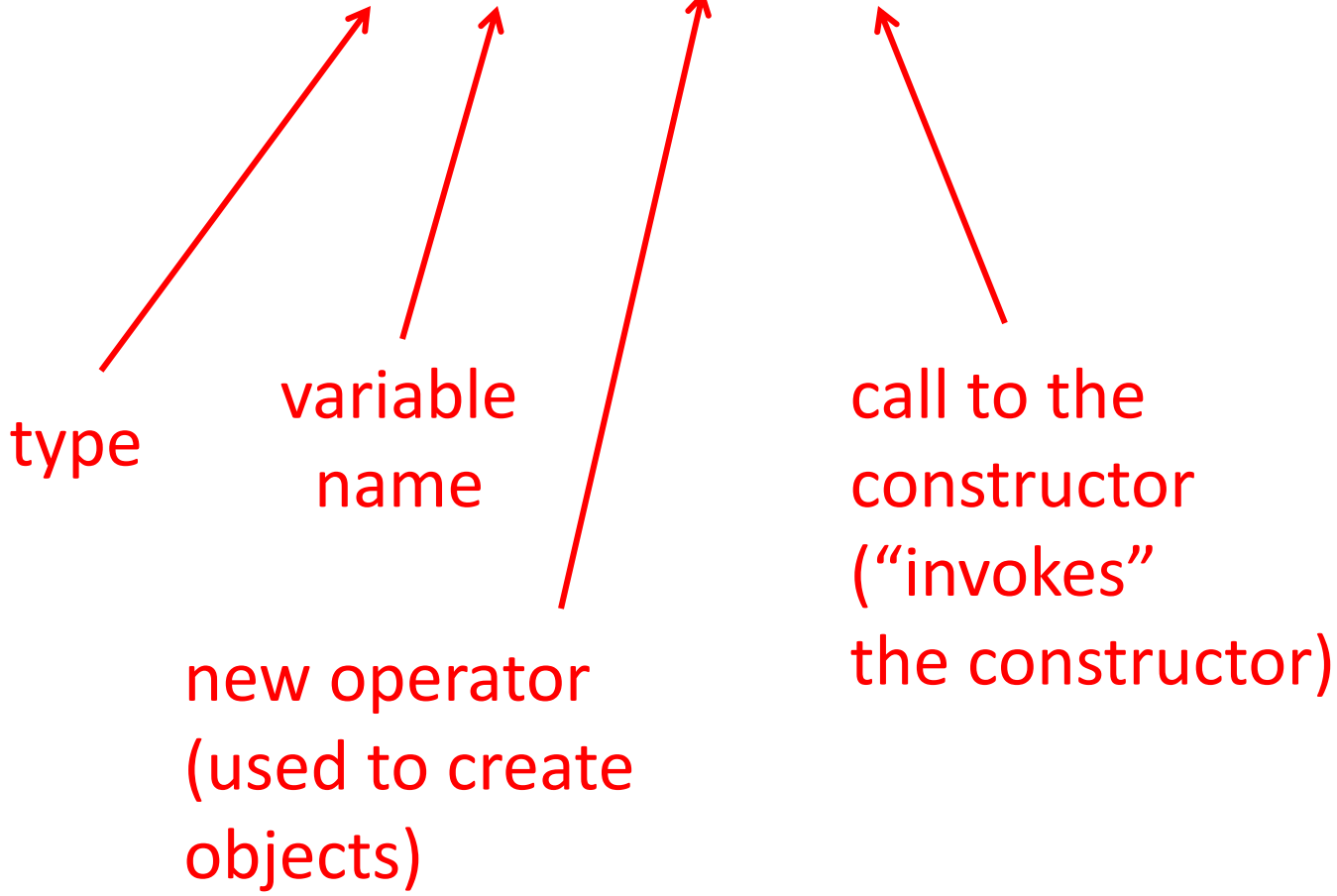
- To create an object, use the **new** keyword and the constructor.
- This is called *instantiating the object* or *creating an instance of* the class.
- **new** does three things:
  - allocates the necessary memory for the object
  - executes the constructor
  - returns the address of (reference to) the object so it can be stored in the variable

Examples: Creating Objects

- Scanner scan = new Scanner(System.in);
  - scan is an *instance of* the Scanner class
- Die d1 = new Die();
  - d1 is an *instance of* the Die class

Creating Objects

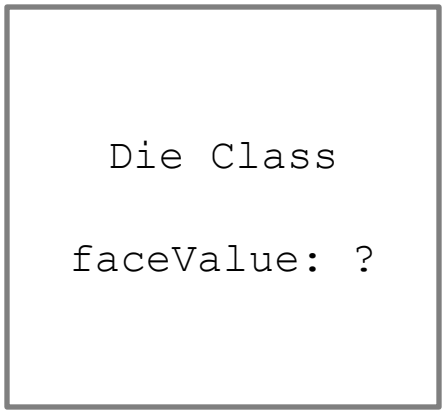
```
Die d1 = new Die();
```



Creating Objects

Die d1 = new Die();

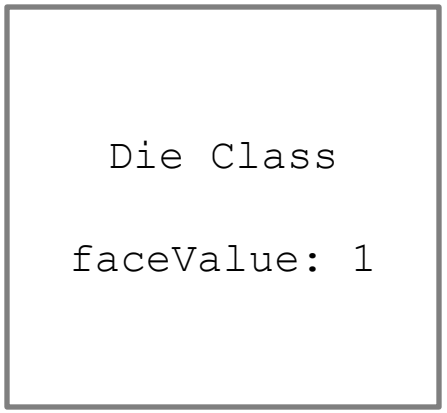
- **new** does three things:
  - allocates the necessary memory for the object
  - executes the constructor
  - returns the address of (reference to) the object so it can be stored in the variable



Creating Objects

Die d1 = new Die();

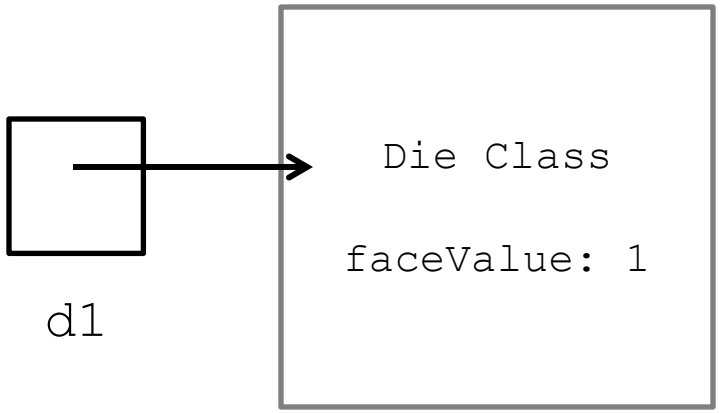
- **new** does three things:
  - allocates the necessary memory for the object
  - executes the constructor
  - returns the address of (reference to) the object so it can be stored in the variable



Creating Objects

Die d1 = new Die();

- **new** does three things:
  - allocates the necessary memory for the object
  - executes the constructor
  - returns the address of (reference to) the object so it can be stored in the variable



Practice

- Create two instances of the Die class.
- Create AudioItem instances.

INVOKING METHODS

Method Invocation

- Invoking a method is like asking an object to "do something."
- When invoked, some methods return a value.
  - The returned value can be saved, used, or ignored.
- void methods do not return a value.

Method Invocation

- To invoke a method, specify:
  - invoking object (or class) followed by dot operator
  - method name
  - actual parameters
- Examples:

```
d1.roll();
d1.setFaceValue(6);
```

Method Invocation

- To invoke a method, specify:
  - invoking object followed by dot operator
  - method name
  - actual parameters
- Examples:

```
d1.roll();
d1.setFaceValue(6);
```

actual parameter

Method Invocation

- We will deal with these special cases later, but I include them here now just for accuracy!

- If the method invoked is in the same class, you only need the method name.
  - You can omit the invoking object.
  - Or you can use `this` as the invoking object.
- If the method is static, you invoke it with the name of the class, rather than with an object.

Declaration vs. Invocation

- Declaring a method specifies what happens when it runs.
- Invoking a method actually *makes* it run.

Practice

- Invoke methods on the Die objects.
- Invoke methods on the AudioItem objects.

- Write a Dice class to represent a pair of dice.

FLOW OF CONTROL

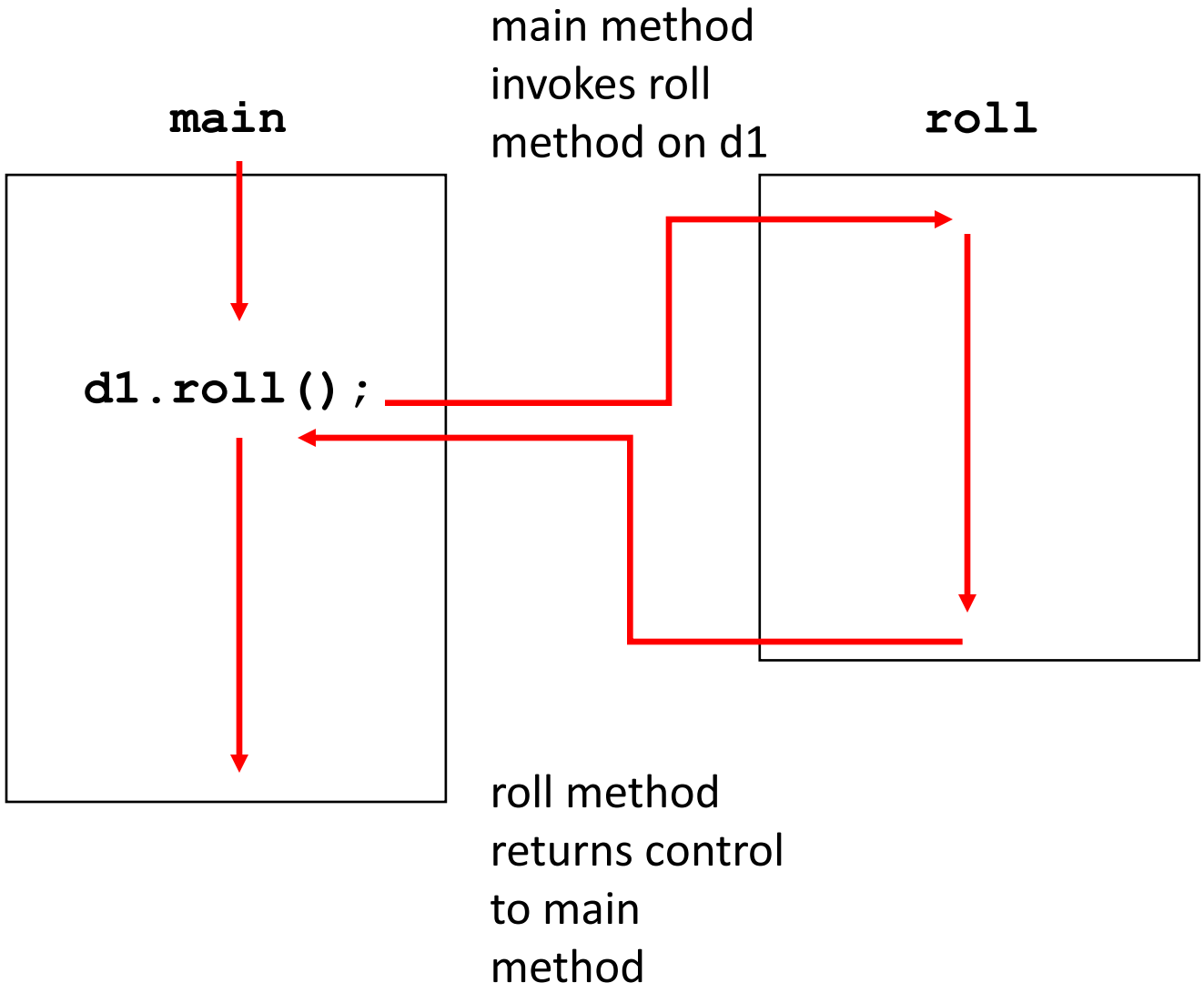
Flow of Control

- Java executes statements in order, starting with the first statement in the main method.
- When a statement contains a method invokation, the flow of control jumps to that method and executes that method line by line.
  - You can think of the original method as being *put on pause* until the invoked method completes.

Flow of Control

- An invoked method is complete when you reach:
  1. a `return` statement or
  2. the end of the method
- When the invoked method is complete, control returns to where the method was called and continues.

Flow of Control Example



INVOKING METHODS WITH PARAMETERS

Parameters

- Formal parameters are defined in the method header
  - They last as long as the method lasts.
  - When the method is over, these parameters are gone!
- Actual parameters are the values sent when the method is invoked.
- When a method is invoked, it's as if there is assignment statement:

```
formalParam = actualParam;
```

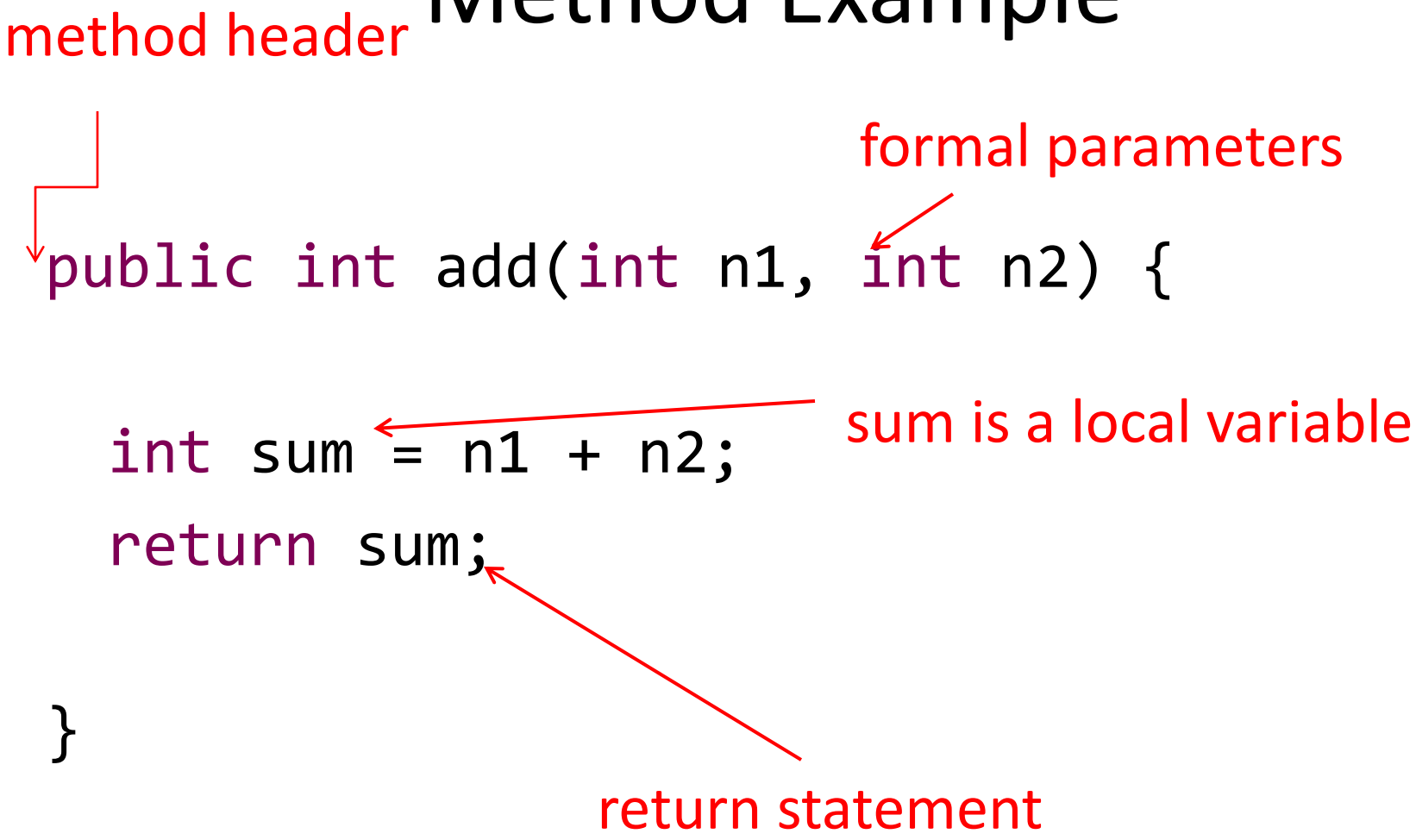
Method Example

```
public int add(int n1, int n2) {

  int sum = n1 + n2;
  return sum;

}
```

Method Example



Method Example

```
public static int add(int n1, int n2) {
  int sum = n1 + n2;
  return sum;
}
public static void main(String[] args) {
  int num = 1;
  int result = add(num, 2);
}
```

formal parameters
(n1 and n2)

actual parameters
(num and 2)

Method Example (cont.)

- When calc is invoked, it is as if we have the statements:

```
n1 = num;
n2 = 2;
```

Passing Parameters

- When a method is invoked, it's as if there is assignment statement:

```
formalParam = actualParam;
```

- But remember the difference between primitives and objects.
  - For primitives, it's the actual value that is stored in memory.
  - For objects, it's a reference to a memory location.
- We'll revisit this later… but for now, keep it in mind!

Variables Revisited

	What It Is	When to Use It	Where It Can Be Used
instance data variable	declared inside the class, outside of any method	when it describes an object	anywhere in the class
local variable	declared inside of a method	when you only need the variable for the method	inside the method only; garbage collected when the method ends
formal parameter	listed in the method header	when you need input to accomplish a task	inside the method only; garbage collected when the method ends
actual parameter	included in the method invokation	when you need to send input to a method	

METHOD OVERLOADING

Method Overloading

- *Method overloading* is the process of giving a single method name multiple definitions
- If a method is overloaded, the name alone is not sufficient to determine which method is being called
- Rather, we need the entire *signature*, including:
  - The number of parameters
  - The type of parameters
  - The order of the parameters

Method Overloading (continued)

- The `println` method is overloaded:

```
public void println  (String s)
public void println  (int i)
public void println  (double d)
```

- The following lines actually invoke different versions of the `println` method:

```
System.out.println("Hello");
System.out.println(35);
System.out.println(42.5);
```

Method Overloading (continued)

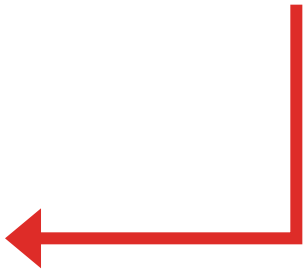
- The compiler determines which method to invoke by analyzing the parameters
- The return type is *not* part of the signature
  - Overloaded methods **cannot** differ only by return type
- It is common to overload constructors
  - This provides multiple ways to initialize a new object

Method Overloading (continued)

```
double tryMe(int x) {
    return x + 0.375;
}
```

```
double tryMe(int x, double y) {
    return x*y;
}
```

Compiler Error!!!



```
int tryMe(int x) {
    return (int) x;
}
```

When to Use Method Overloading

- Use method overloading only when you have multiple methods that take different parameters but do the *same* thing.
  - You often want the client to think it's the same method (e.g., println).
- You should not overload methods if two methods perform different tasks.

Practice

- Overload the roll method.

JAVA-PROVIDED CLASSES

Class Libraries

- A *class library* is a collection of classes
- The *Java standard class library* is often called the Java API (Application Programming Interface)
- The Java API is included in any development environment (like Eclipse)
- We rely heavily on many classes, but they are not part of the language.
  - System
  - Scanner
  - String

Packages

- Classes libraries are organized into packages
  - By convention, packages are lowercase
- To use a class, you typically:
  - import the class

```
import java.util.Scanner;
```

  - import the package (all classes)

```
import java.util.*;
```

Packages (cont.)

- As a general rule of thumb, if you are using more than one class in a package, you can import the entire package.
- If you don’t import it and don’t use a fully qualified name, you will get a compiler error.
  - Java doesn’t know where the Scanner class is if you don’t tell it!

The `java.lang` Package

- All classes of the `java.lang` package are imported automatically into all programs.
- It's as if all programs contain:

```
import java.lang.*;
```

- This is why we don't have to import anything to use `System` or `String`

The `Random` Class

- The Random class allows you to generate *pseudorandom* numbers.
- Random is in the `java.util` package.
- To create an object of the `Random` class:

```
Random generator = new Random();
```

The `Random` Class (cont.)

- `generator.nextInt()`
  - generates a random integer
- `generator.nextInt(n)`
  - generates a random integer between 0 (inclusive) and n (exclusive)
  - n options to be returned
- `generator.nextDouble()`
  - generates a random double between 0.0 (inclusive) and 1.0 (exclusive)
- `generator.nextBoolean()`
  - generates a random boolean

The `Random` Class (cont.)

```
Random generator = new Random();
int n = generator.nextInt(5);
// generates a random number
//    between 0 and 4 (inclusive)
// 5 options: 0, 1, 2, 3, 4
```

The `Random` Class (cont.)

```
n = generator.nextInt(5) + 4;
// generates a random number
//    between 4 and 8 (inclusive)
// 5 options: 4, 5, 6, 7, 8
```

The nextInt method

- The parameter is the number of options
- The number added on is how many numbers above (or below) zero will begin the range

The `Random` Class (cont.)

- You can also multiply the random number by an integer to create numbers that are separated by that integer.

```
n = generator.nextInt(5) * 2;
   // generates a random number
   // in the set {0, 2, 4, 6, 8}
```

Examples

- How would you generate:
  - a number between -5 and 5 (inclusive)
  - a number between -5 and 5 (exclusive)
  - an even number between 0 and 20 (inclusive)
  - an odd number between 0 and 20
  - a two-digit integer
  - a three-digit integer

Practice

- Update the roll method in the Die class.

The `Math` Class

- The `Math` class provides methods that perform various mathematical functions, including:
  - Absolute value: `Math.abs(num)`
  - Square root: `Math.sqrt(d)`
  - Exponentiation: `Math.pow(num, pow)`
  - Trig functions: `Math.sin(d)`, etc.
  - Constants: `Math.PI`, `Math.E`

The `Math` Class (cont.)

- The methods of the `Math` class are *static* methods (also called *class* methods)
  - You don't create objects to invoke these methods.
  - Invoke the methods through the class name.
- The `Math` class is in the `java.lang` package, so you do not need to import it.

Practice

- Write code to calculate the following based on user's input:
  - area of a circle

    $A = \pi r^2$ $C = 2 \pi r$
  - the distance between two points

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Wrapper Classes

- The `java.lang` class contains a *wrapper class* for each primitive type

Primitive Type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

Primitives and Objects

- Wrapper classes can be used when you have a primitive variable but need an object or vice versa.

```
Integer age = new Integer(40);
Boolean status = new Boolean(true);
Character c = new Character('c');
```

Autoboxing

- Primitive values will be automatically converted to a corresponding wrapper object and vice versa

```
int num = 42;
Integer objectInt = num;

Integer objectInt2 = 21;
int num2 = objectInt2;
```

Wrapper Class Constants

Integer.MIN_VALUE;

Integer.MAX_VALUE;

Double.SIZE;

Parsing Methods

- Each wrapper class provides a method to *parse* a String and convert it to a primitive type
  - Very helpful when reading input from users!

```
String intString = “123”;
String doubString = “142.3”;
int n = Integer.parseInt(intString);
double d = Double.parseDouble(doubString);
```

`toString` Methods

- Each wrapper also provides a `toString` method to convert a primitive to a String
  - Very helpful when sending output to users!

```
int n = 45;
double d = 2.3;
String nString = Integer.toString(n);
String dString = Double.toString(d);
```

THE STRING CLASS

Strings

- *String*: A sequence of characters surrounded by double quotes (also called a *string literal*)
  - `"This is a string literal."`
  - `"123 Main Street!"`
  - `"X"`
  - `""`
- `String` is a class
- String variables are objects of the `String` class

Creating Strings

- Using the constructor
  - String s = new String(“hello”);
- Shortcut!
  - String t = “goodbye”;

String Methods

- String objects are special objects called *immutable*.
  - They cannot be changed
- To change a `String` reference, create a new `String` and point your reference at that new `String`

length

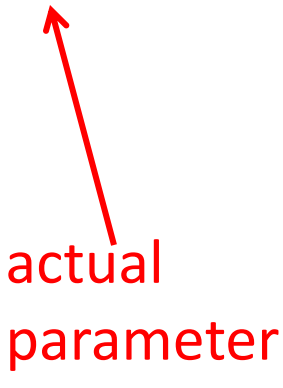
- public int length()
- Returns the number of characters in the String

```
String s = "Hello there!";
int n = s.length();
// n holds 12
```

Concatenation

- public String concat(String str)
  - Returns a new string which is the *invoking object* concatenated with the *actual parameter*
  - Does **not** change the invoking object

```
String firstName = "Jessica";
String lastName = "Masters";
String fullName = firstName.concat(lastName);
// same as saying firstName + lastName
```



Concatenation (cont.)

- You can assign the new String back to the original variable

```
String s1 = “Hello”;
String s2 = “ There”;
s1 = s1.concat(s2);

// same as s1 = s1+ s2
// same as s1 += s2
```

Replacing Characters

- public String replace (char oldChar, char newChar)

```
String s1 = “Hello”;
String s2 = s1.replace(‘e’,’a’);
// s1 still holds “Hello”
// s2 holds “Hallo”
```

Case

- public String toLowerCase()
- public String toUpperCase()

```
String s1 = “Hello”;
String s2 = s1.toUpperCase();
// s1 still holds “Hello”
// s2 holds “HELLO”
```

Substrings

- Each character in a string has an index
  - Indices begin at zero
- public String substring(int offset, int endIndex)
  - Returned substring will start at `offset`
  - Returned substring will end at (`endIndex` – 1)
  - Length of returned substring is (`endIndex` – `offset`)

Substrings (cont.)

```
String greeting = "Good night";
//  G   o   o   d       n   i   g  h  t
//  0   1   2   3   4   5   6   7  8  9

String shorter = greeting.substring(3, 7);
// holds "d ni"
// length = 7-3 = 4

shorter = greeting.substring(0,1);
// holds "G"
// length = 1-0 = 1
```

String Summary

- String are immutable!
  - Invoking a method returns a new String- it does not change the invoking object.
- Strings are indexed starting at position 0.

Practice

- Write a text-only program to generate a username based on a user's first name, last name, and a random number. The username should consist of the first letter of the user's first name, up to six letters of the user's last name, and a two-digit random number (all lower case).