CSC 1051 Algorithms & Data Structures I

Classes, Objects, Selection & More Repetition

OBJECTS & CLASSES

Objects and Classes

An object is a fundamental program component

It represents something in terms of attributes and behaviors

Potential attributes of a car:

make, model, color, current speed, current direction

Potential behaviors of a car:

accelerate, stop, change gears, turn

Which attributes and behaviors are actually represented depends on what the system does

A car object in a racing game would be different than a car object in a dealer inventory system

Objects and Classes

An object is defined by a class

A class is:

- the type of the object
- the pattern from which it is created

You might write a class called Car and then create many Car objects using it

The word class comes from the word *classification*

A class represents a group of similar objects

An object is an instance of a class

Each object has its own instance data that represents its attributes

Objects and Classes

An object has three properties

identity – the way you specify an individual object

A reference variable is the object's identity in a program

state – the values of its instance data

One car: green Honda CRV heading northeast at 55 MPH

Another car: black Ford Focus heading south at 40 MPH

behavior – the list of services an object can perform

The methods you can invoke on an object define its behavior (also known as its public interface)

Objects and Classes

A class is like the blueprint of an object

The concept

The realization

You can't live in a blueprint, but it defines the house

Houses made from the same blueprint are the same type of house, but have room for different furniture and families

Objects and Classes

A class defines the types of data an object will have and determines how that data will be organized

But it doesn't reserve any memory space for it until an object is created

Each object has its own memory space and therefore its own state

The class contains the code for the methods that define an object's behavior

But the methods are called through a particular object, which determines which data is used and updated

Creating Objects

The java.awt.Point class represents a two-dimensional x, y coordinate

To create a Point object, we use the new operator

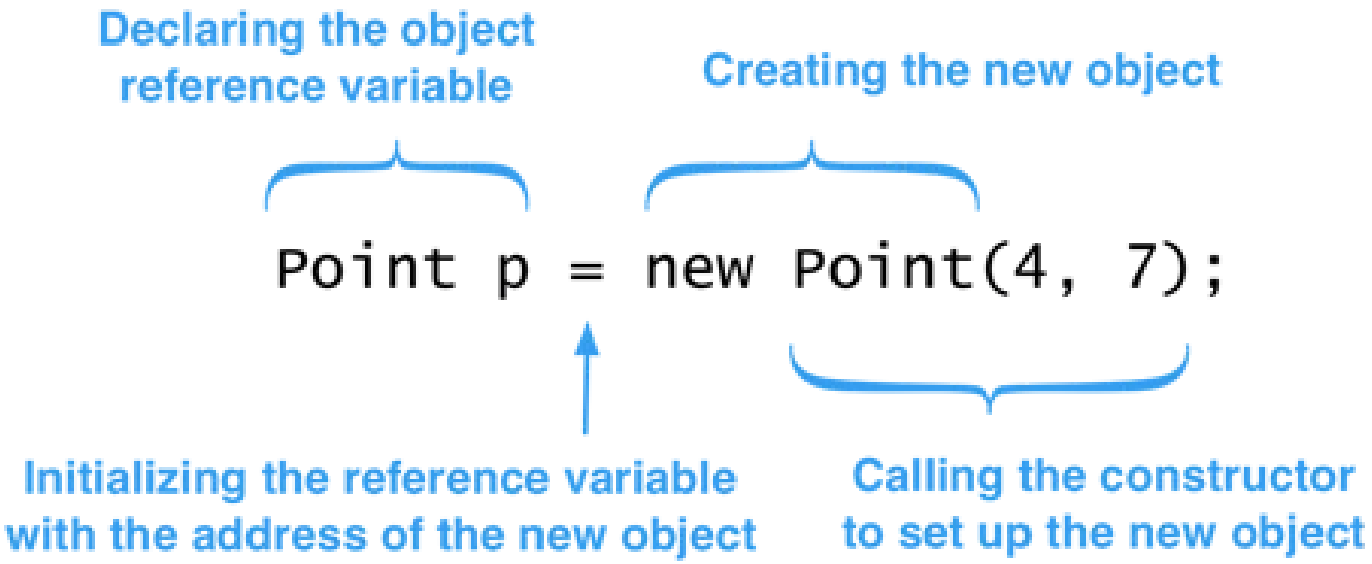
```
Point p = new Point(4, 7);
```

A call to the object's constructor sets up the new object

A constructor is like a method that initializes instance data and whatever else is necessary to get the object ready to use

A constructor has the same name as the class

Creating Objects

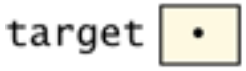


These activities are often combined this way, but they don't have to be

Creating Objects

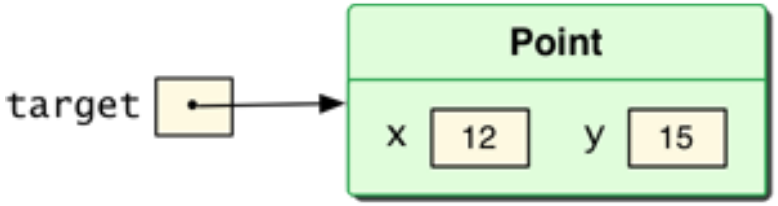
If a reference variable is null, it doesn't refer to any object

```
Point target = null;
```



Later on, it can be assigned an object

```
target = new Point(12, 15);
```



Creating Objects

Beware the null reference!

Following a null reference will cause a NullPointerException to be thrown

```
Point nowhere = null;
double x = nowhere.getX();
```

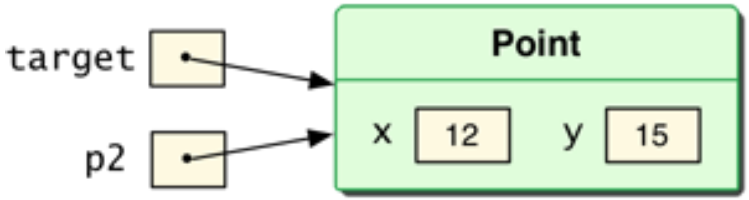
When in doubt, check to see if a reference is null

```
if (myPoint != null)
    x = myPoint.getX();
```

Creating Objects

Multiple references can refer to the same object

```
Point p2 = target;
```



They are sometimes referred to as aliases of each other

Changes made to one affect the other because they are the same object

Creating Objects

You can create a String object with the new operator

```
String name = new String("George R. R. Martin");
```

But a double quoted string literal already represents a String object, so you can just do this:

```
String name = "George R. R. Martin";
```

And even this:

```
System.out.println("alakazam!".toUpperCase());
```

```
ALAKAZAM!
```

Example: Palindromes

A palindrome is a string that reads the same forwards and backwards

radar
kayak
deified
drab bard
able was I ere I saw elba

Goal: determine if a string is a palindrome

For now, spaces, punctuation, and differences in case all affect the conclusion

The import Statement

Classes in the Java API is part of a particular package

String is in the java.lang package

Random is in the java.util package

A package is a group of related classes

A class can always be referred to by its fully qualified name

java.util.Random

Package organization allows two classes to have the same name, such as

java.util.Timer and java.awt.Timer

The import Statement

Using the fully qualified name is not always convenient:

```
java.util.Random gen = new java.util.Random();
```

An alternative is to import the class

Import statements go above the class that uses them

```
import java.util.Random;
```

Then the simple name can be used throughout the program:

```
Random gen = new Random();
```

The import Statement

Classes from the java.lang package are automatically imported

That's why we don't import classes like System or String

If you're going to use multiple classes from a package, you can use the * wildcard character:

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

THE JAVA API

The Math Class

Numeric expressions often rely on methods of the Math class

The Math class is part of the java.lang package, so does not have to be imported

You don't (can't) make an object of type Math

The methods in the Math class are static, so are called through the class

```
double result = Math.abs(num) + Math.pow(count, 3);
```

The Math Class

The Math class also contains two useful public constants

π →

Constant	Value
Math.PI	3.141592653589793
Math.E	2.718281828459045

← Base of the natural log

```
double circumference = 2 * Math.PI * radius;
```

The Math Class

Absolute value, minimum and maximum

```
Math.abs(5)                    5
Math.abs(-7.34)             7.34

Math.min(35, 12)              12
Math.min(-18.5, -7)        -18.5

Math.max(28, -5)              28
Math.max(-7.4, -12.2)       -7.4
```

The Math Class

Rounding, floor (nearest integer less than), and ceiling (nearest integer greater than)

```
Math.round(15.33)        15
Math.round(15.7)         16
Math.round(7.5)           8

Math.floor(8.2)         8.0
Math.floor(8.9)         8.0
Math.floor(-5.2)       -6.0

Math.ceil(8.2)          9.0
Math.ceil(8.9)          9.0
Math.ceil(-5.2)        -5.0
```

The Math Class

Square root and exponentiation (raising to a power)

```
Math.sqrt(25)                          5.0
Math.sqrt(30)           5.477225575051661
Math.sqrt(123.45)      11.110805551354051

Math.pow(2, 3)                         8.0
Math.pow(2.5, 3.5)     24.705294220065465
Math.pow(-1.5, 3)                   -3.375
```

The Math Class

The Math.random method returns a random number in the range 0.0 (inclusive) to 1.0 (exclusive)

```
Math.random()        0.7264869439957039
Math.random()     0.42153058405914756
```

The Random class offers other methods for creating random numbers

Random Numbers

A random number generator is an object that produces a stream of pseudorandom numbers

They are based on a seed value that factors into a set of calculations

To the user, they certainly appear random

Two mechanisms for generating random numbers in Java:

Random class

Math.random method

Random Numbers

The Random class has several methods for generating random numbers

The nextInt method accepts an argument N and returns an integer in the range 0 to N-1

```
Random generator = new Random();
int num = generator.nextInt(10);
```

The variable num now contains a single integer between 0 and 9, inclusive

There's an equal probability of getting any value in that range

Random Numbers

The argument to nextInt is called a scaling factor because it determines the size of the range of values

A shift value can also be added to shift the starting point of the range

```
num = generator.nextInt(50) + 1;
```

The call to nextInt returns a value between 0 and 49, which is then shifted into the range 1 to 50

Random Numbers

In general, a scale factor of X and a shift value of Y produces an integer in the range Y to X + Y – 1

Expression	Range
`generator.nextInt(100)`	0 to 99
`generator.nextInt(256)`	0 to 255
`generator.nextInt(6) + 1`	1 to 6
`generator.nextInt(20) + 100`	100 to 119
`generator.nextInt(50) - 10`	-10 to 39
`generator.nextInt(10) - 50`	-50 to -41

Random Numbers

The seed value (a long integer) can be set for a Random object by passing it into the constructor

```
Random generator = new Random(54321);
```

The Random class also has a setSeed method

The seed value determines exactly the stream of numbers that will be produced

Random Numbers

The Math.random method returns a random double value in the range 0.0 to 1.0 (excluding 1.0)

It is essentially the same as the nextDouble method of the Random class, except you don't have to create an object first

```
System.out.println(Math.random());
```

```
0.7251182764665118
```

The seed cannot be explicitly set for the Math.random method (it uses the system time as the seed)

Random Numbers

A random floating-point value can be converted to an integer in a particular range with a calculation

```
int num = (int) (Math.random() * 10);
```

Multiplying the random value by 10 (the scaling factor), then casting it to an int results in an integer in the range 0 to 9

Adding a shift value shifts the range

To produce a random number in the range 6 to 35:

```
num = (int) (Math.random() * 30) + 6;
```

SELECTION & MORE REPETITION

Example: The High-Low Game

The user guesses a predetermined number in as few guesses as possible

The set up:

```
Scanner in = new Scanner(System.in);
Random generator = new Random();

int target = generator.nextInt(100) + 1;
int guess = -999;  // initial value out of range
int count = 0;

System.out.println("I've chosen a number " +
    "between 1 and 100.");
```

The do-while Statement

A do-while statement is another Java repetition statement

It uses the keywords do and while, with the condition shown after the body of the loop

The condition governing the loop is not evaluated until after the body is executed

Therefore, the body of a do-while loop is executed at least once (unlike the while loop)

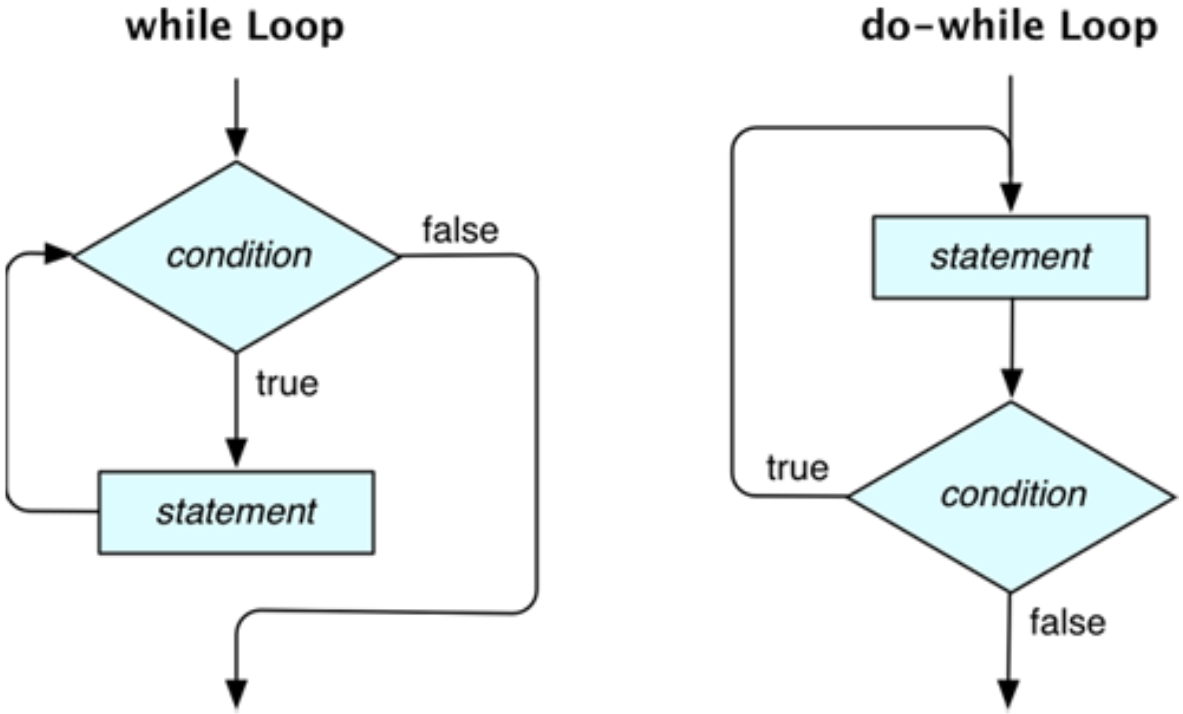
The do-while Statement

```
int num = 0;

do
{
    num++;
    System.out.println(num);
}
while (num < 5);

System.out.println("Now here.");
```

```
1
2
3
4
5
Now here.
```

The do-while Statement

Comparing the while loop and the do-while loop:



The do-while Statement

Using a do-while loop for input validation

```
Scanner in = new Scanner(System.in);
double num;

do
{
    System.out.print("Enter a number greater than 100: ");
    num = in.nextDouble();
    if (num <= 100)
        System.out.println("Invalid number.");
}
while (num <= 100);

System.out.println("Moving on...");
```

The do-while Statement

The do-while loop is simply not used that often in production code

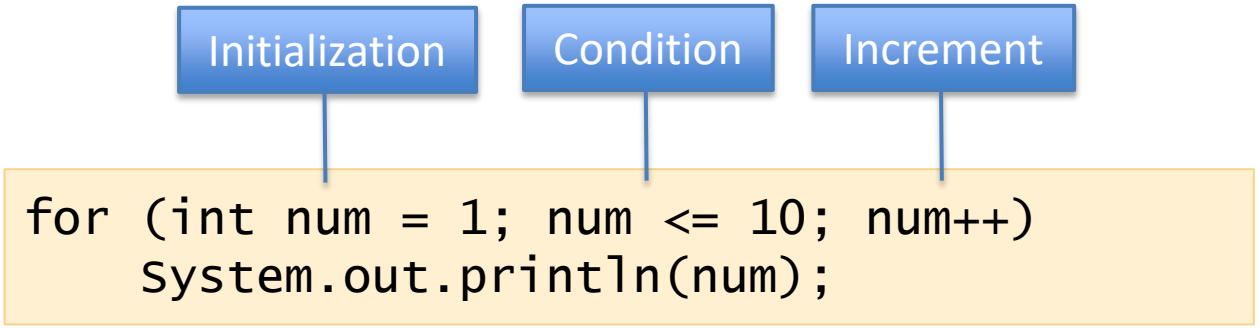
- It's often just as easy to use a while loop
- The use of the while keyword can be misread

Many developers simply avoid using it

For Statement

A for statement is a loop that works well when you know or can calculate how many iterations need to be performed

The for loop header contains three sections separated by semicolons



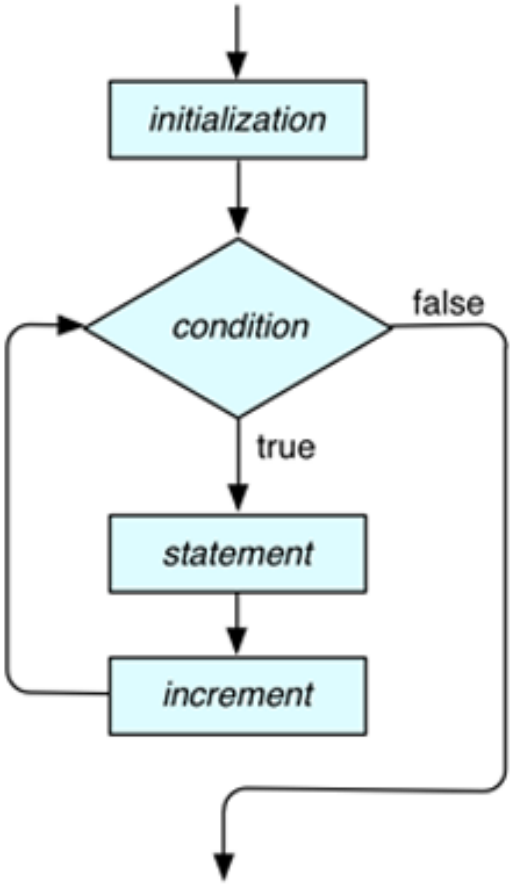
The control variable is often declared in the initialization section, but doesn't have to be

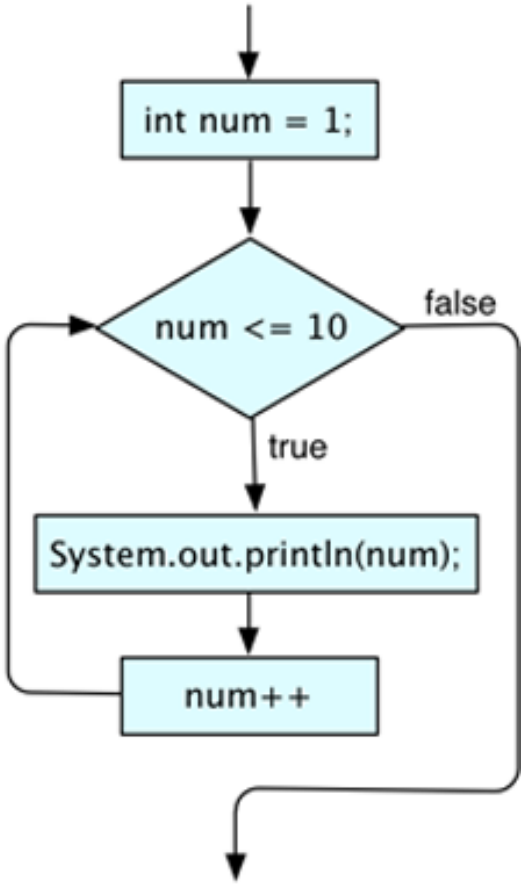
For Statement

```
for (int num = 1; num <= 10; num++)
    System.out.println(num);

System.out.println("Now here.");
```

```
1
2
3
4
5
6
7
8
9
10
Now here.
```

For Statement





For Statement

The for statement is compact and often convenient

Equivalent code could always be written as a while loop

```
for (int num = 1; num <= 10; num++)
    System.out.println(num);
```

```
int num = 1;
while (num <= 10)
{
    System.out.println(num);
    num++;
}
```

For Statement

The increment section does not have to increment

```
for (int i = 20; i > 0; i--)
    System.out.print(i + " ");

System.out.println();
System.out.println("Now here.");
```

```
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Now here.
```

For Statement

Printing the powers of two less than 1000:

```
for (int num = 2; num < 1000; num *= 2)
    System.out.print(num + " ");

System.out.println();
System.out.println("Now here.");
```

```
2 4 8 16 32 64 128 256 512
Now here.
```

For Statement

Printing a triangle made of asterisks:

```
for (int line = 1; line <= 7; line++)
{
    for (int asterisk = 1; asterisk <= line; asterisk++)
        System.out.print("*");

    System.out.println();
}
```

The outer loop's control variable

```
*
**
***
****
*****
******
*******
```

Conditional Expressions

A conditional expression evaluates a boolean condition and returns one of two results

So instead of writing this:

```
if (y > 0)
    x = 1;
else
    x = -1;
```

You could perform the same task with one assignment:

```
x = (y > 0) ? 1 : -1;
```

Conditional Expressions

The conditional operator (?:) is used like this:

condition ? *expression1* : *expression2*

First, the boolean condition is evaluated

If it's true, then the result of *expression1* is returned

If not, the result of *expression2* is returned

It's a ternary operator (three operands)

It's the only operator in Java whose characters can be separated

Conditional Expressions

This example assigns the larger of x1 and x2 to the variable bigger:

```
bigger = (x1 > x2) ? x1 : x2;
```

This statement prints the appropriate word depending on the value of num:

```
System.out.println(num + " " (num == 1 ? "box" : "boxes"));
```

```
1 box
```

```
5 boxes
```

You usually don't need parentheses

Switch Statement

Like an if statement a switch statement lets you determine which statement is executed next

The expression in the switch header is evaluated and processing continues with the first matching case

If a break statement is executed, processing jumps to the statement following the switch

An optional default case can be specified in the event no case matches

Switch Statement

```
switch (dayNum)
{
    case 1:
        day = "Sunday";
        break;
    case 2:
        day = "Monday";
        break;
    case 3:
        day = "Tuesday";
        break;
  ...
    case 7:
        day = "Saturday";
        break;
    default:
        System.out.println("Invalid.");
}
```

Switch Statement

A switch statement is restricted in various ways

- Case values must be constants – they cannot be variables or expressions
- The switch expression must evaluate to an int, char, enumeration constant, or (as of Java 7) a String
- A switch only tests equality – you cannot make relational comparisons such as less than (<)

Switch Statement

If a break statement is not used to terminate a case, processing continues into the next case

```
switch (letter)
{
    case 'A':
        System.out.println("here");
    case 'B':
        System.out.println("there");
        break;
    case 'C':
  ...
}
```

No break statement

Switch Statement

Sometimes, this fall through behavior is helpful, but should be documented

```
switch (dayNum)
{
    case 2:  // fall through
    case 4:  // fall through
    case 6:
        System.out.println("M/W/F Schedule");
        break;
    case 3:  // fall through
    case 5:
        System.out.println("T/Th Schedule");
        break;
    default:
        System.out.println("Weekends rock!");
}
```

Switch Statement

The functionality of a switch can always be accomplished with a nested if statement

```
switch (salesCategory)
{
    case HOURLY:
        commission = 0.0;
        break;
    case COMMISSION:
        commission = 0.15;
        break;
    case OPERATOR:
        commission = 0.0;
        limit = 20;
        break;
    case MANAGER:
        commission = 0.25;
        break;
    default:
        System.out.println("Invalid category");
}
```

```
if (salesCateogry == HOURLY)
{
    commission = 0.0;
}
else if (salesCateogry == COMMISSION)
{
    commission = 0.15;
}
else if (salesCateogry == OPERATOR)
{
    commission = 0.0;
    limit = 20;
}
else if (salesCateogry == MANAGER)
{
    commission = 0.25;
}
else
{
    System.out.println("Invalid category");
}
```