Constructors

CMSC 202

Object Creation

- Objects are created by using the operator ***new*** in statements such as…

```
Car c = new Car();
```

- The following expression invokes a special kind of method known as a ***constructor***…

```
new Car();
```

- Constructors are used to
  - Create objects and
  - Initialize the instance variables

Constructors

- A constructor
  - Has the same name as the class it constructs
  - Has no return type (not even void)
- If the class implementer does not define any constructors, the Java compiler automatically creates a constructor that has no parameters
- Constructors may be (and often are) overloaded

The (Almost) Finished Car Class

```
public class Car {
   private int numLiters;
   private int horsepower;
   private int numDoors;
   private int year;
   private String color;
   private String model;
   private String make;
   private String vin;

   // a constructor that accepts all state attributes
   public Car(String vin, String color, String make, String model,
              int numLiters, int horsepower, int numDoors, int year) {
        this.vin = vin;
        this.model = model;
        this.make = make;
        this.color = color;
        this.numLiters = numLiters;
        this.horsepower = horsepower;
        this.numDoors = numDoors;
        setYear(year);
   }
```

Car Class (continued)

```
    // a constructor that uses parameters and default state values
    public Car(String vin, int year, String make, String model) {
     this.vin = vin;
     this.make = make;
     this.model = model;
     setYear(year);
     numLiters = 2;
     horsepower = 200;
     color = "blue";
     numDoors = 2;
    }
    // a default constructor
    public Car() {
       vin = "1234567";
       make = "Ford";
       model = "Focus";
       year = 2011;
       numLiters = 2;
       horsepower = 200;
       color = "blue";
       numDoors = 2;
    }
    // ...
}
```

Using Car Constructors

```
public static void main(String args[]) {
        Car defaultCar = new Car();
        System.out.println("My Car: " + defaultCar);

        Car chevy = new Car("9431a", 2000, "Chevy", "Cavalier");
        System.out.println("Chevy Car: " + chevy);

        Car dodge = new Car("8888", "orange", "Dodge", "Viper",
                        5, 400, 2, 1996);
        System.out.println("Dodge Car: " + dodge);
}
```

defaultCar	chevy	dodge
vin: “1234567”	vin: “9431a”	vin: “8888”
make: “Ford”	make: “Chevy”	make: “Dodge”
model: “Focus”	model: “Cavalier”	model: “Viper”
year: 2011	year: 2000	year: 1996
numLiters: 2	numLiters: 2	numLiters: 5
horsepower: 200	horsepower: 200	horsepower: 400
color: “blue”	color: “blue”	color: “orange”
numDoors: 2	numDoors: 2	numDoors: 2

this( ) Constructor

- When several alternative constructors are written for a class, we reuse code by calling one constructor from another
- The called constructor is named **`this()`**

Copy Constructor

- Another common form of a constructor is called a ***copy constructor***
- A copy constructor takes a single argument that is the same type as the class itself and creates a copy of it…

```
// copy constructor
public Car(Car otherCar) {
    this(otherCar.vin, otherCar.color, otherCar.make,
         otherCar.model, otherCar.numLiters,
         otherCar.horsepower, otherCar.numDoors,
         otherCar.year);
}
```

Better Car Constructors

```
// a constructor that uses parameters and default state values
public Car(String vin, int year, String make, String model) {
    this(vin, "blue", make, model, 2, 200, 2, year);
}

// a default constructor
public Car() {
    this("1234567", "blue", "Ford", "Focus", 2, 200, 2, 2011);
}

// a constructor that accepts all state attributes
public Car(String vin, String color, String make, String model,
           int numLiters, int horsepower, int numDoors, int year) {
    this.model = model;
    this.vin = vin;
    this.make = make;
    this.color = color;
    this.numLiters = numLiters;
    this.horsepower = horsepower;
    this.numDoors = numDoors;
    setYear(year);
}
```

What Happens in Memory: The Stack and Heap

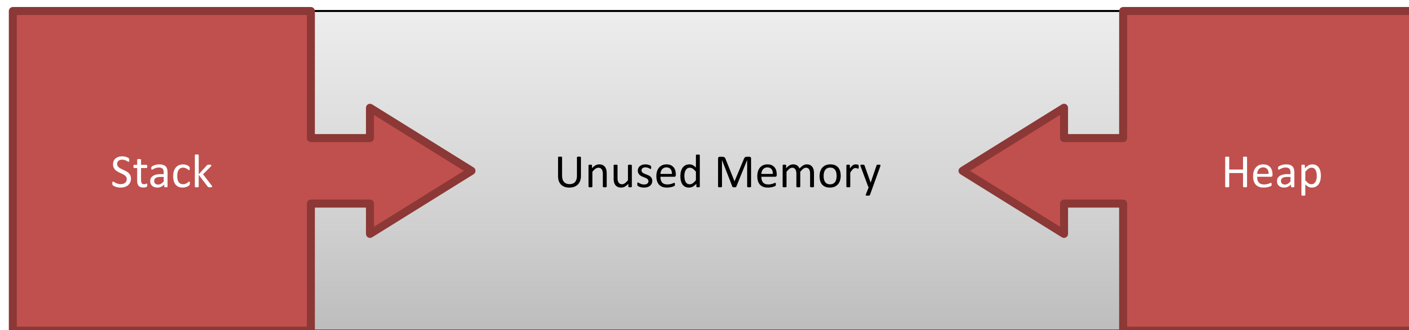
- When your program is running, local variables are stored in an area of memory called the ***stack***.
- A table can be used to illustrate variables stored on the stack:

Variable	Value
x	42
y	3.7

- The rest of memory is known as the ***heap*** and is used for ***dynamically allocated*** "stuff".

Main Memory

- The stack grows and shrinks as needed. (why?)
- The heap also grows and shrinks. (why?)
- Some of memory is unused. (“free”)



Object Creation

- Consider this code that creates two Cars:

```
Car c1, c2;
c1 = new Car("A", 2000, "Ford", "Explorer");
c2 = new Car("B", 2009, "Nissan", "Titan");
```

- Where are these variables and objects located in memory?
- Why do we care?

Objects in Memory

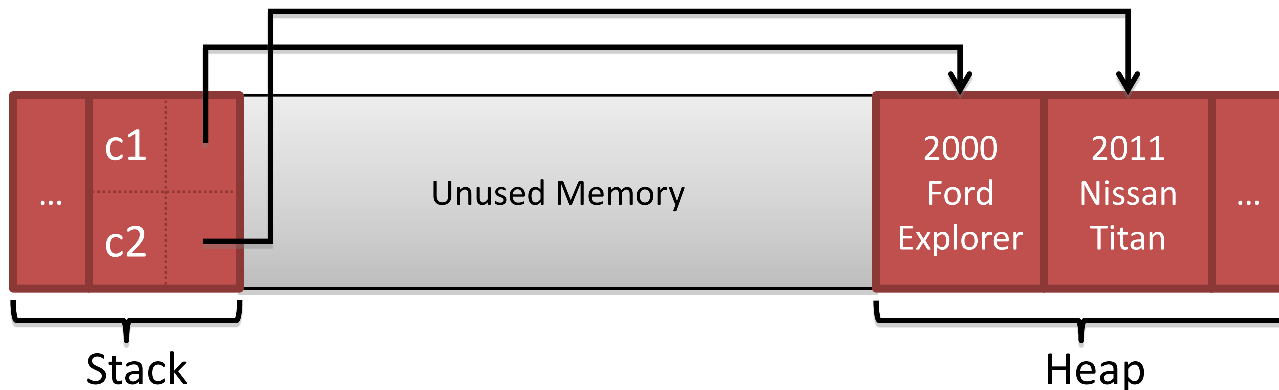
- The following creates two local variables on the ***stack***…

```
Car c1, c2;
```

- Whereas the instantiation of actual objects are created on the ***heap***…

```
c1 = new Car("A", 2000, "Ford", "Explorer");
c2 = new Car("B", 2009, "Nissan", "Titan");
```

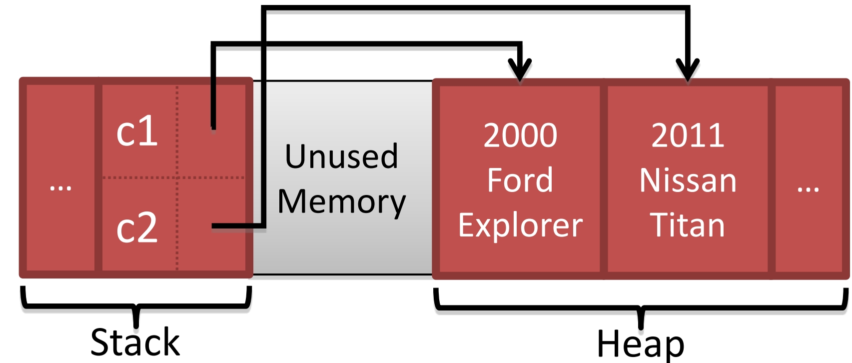
- c1 and c2 contain the ***memory addresses*** of these objects giving us the picture of memory shown below — these are known as ***reference variables***
- Reference variables which do not contain the memory address of any object contain the special value null



Why We Care (1 of 4)

- Given the previous code and corresponding picture of memory…

```
Car c1, c2;
c1 = new Car("A",2000,"Ford","Explorer");
c2 = new Car("B",2009,"Nissan","Titan");
```



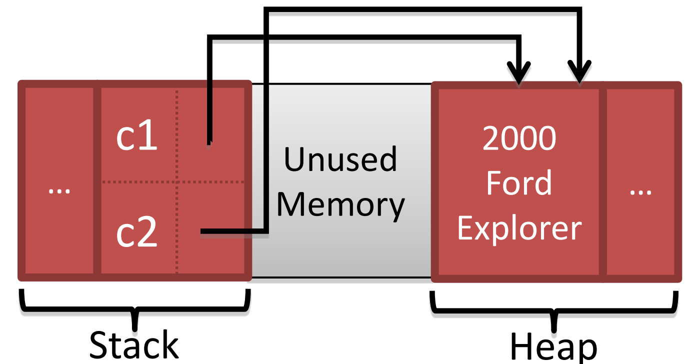
- Consider the expression `c1 == c2`
- Recall that `c1` and `c2` contain the addresses of their respective Car objects. Since the Car objects have different addresses on the heap, `c1 == c2` is ***false***
  - The == operator determines if two reference variables refer to the same object
- So how do we compare Car for equality?
  - Cars (and other objects) should implement a method named equals

```
c1.equals(c2);
```

Why We Care (2 of 4)

- On the other hand, consider this code and corresponding picture of memory

```
Car c1 = new Car("A",2000,"Ford","Explorer");
Car c2 = c1;
```



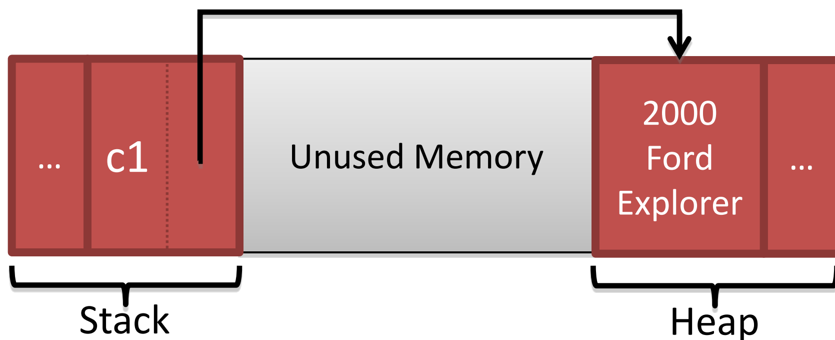
- Now `c1` and `c2` refer to the same Car object. This is known as ***aliasing***, is often unintentional, and can be dangerous. Why?
- If your intent is for `c2` to be a copy of `c1`, then the correct code is…

```
Car c2 = new Car(c1);
```

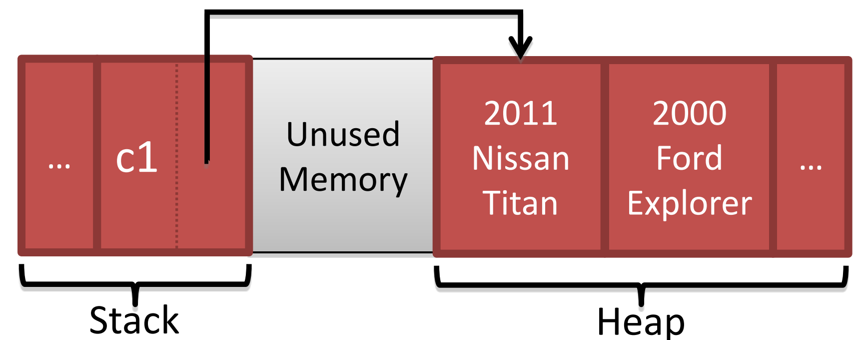
Why We Care (3 of 4)

- Consider this code and the changing picture of memory…

```
Car c1 = new Car("A",2000,"Ford","Explorer");   // line 1
c1 = new Car("B",2011,"Nissan","Titan");        // line 2
```

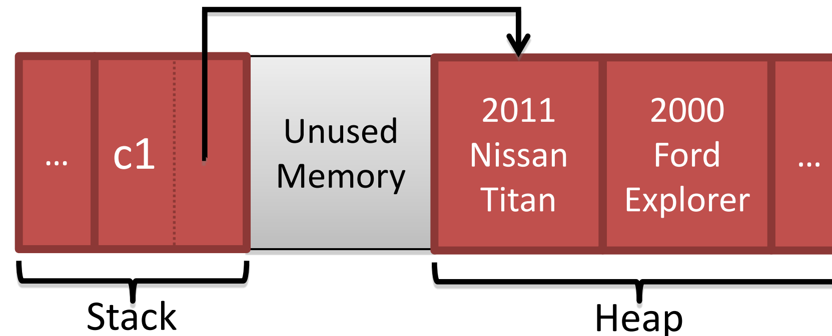


After Line 1



After Line 2

Why We Care (4 of 4)



- As the diagram shows, after line 2 is executed no variable refers to the Car object which contains 2000, “Ford”, “Explorer”
- In C/C++, we’d consider this a ***memory leak***. In C/C++ it’s the ***programmer’s responsibility*** to return dynamically allocated memory back to the free heap. Not so in Java!
- Java has a built-in ***garbage collector***. From time to time Java detects objects that have been ***orphaned*** because no reference variable refers to them. The garbage collector automatically returns the memory for those objects to the free heap.

Arrays of a Class Type

- The base type of an array can be a class type as well as a primitive type.
- This statement creates 20 indexed ***reference variables*** of type `Car`.

```
Car[] carsInParkingGarage = new Car[20];
```

- It ***does not*** create 20 objects of the class `Car`.
- Each of these indexed variables are automatically initialized to `null`.
- Any attempt to reference any of them at this point would result in a ***null pointer exception*** error message.

Variables Review: Primitives vs. References

- Every variable is stored at a location in memory.
- When the variable is a ***primitive type***, the value of the variable is stored in the memory location assigned to the variable.
  - Each primitive type always requires the same amount of memory to store its values.

Variables Review: Primitives vs. References

- When the variable is a ***class type***, only the memory address (or ***reference***) where its object is located is stored in the memory location assigned to the variable (on the stack).
- The object named by the variable is stored in the heap.
- Like primitives, the value of a class variable is a fixed size.
- The object, whose address is stored in the variable, can be of any size.

Class Parameters

- All parameters in Java are ***pass-by-value parameters***.
  - A parameter is a ***local variable*** that is set equal to the value of its argument.
  - Therefore, any change to the value of the parameter cannot change the value of its argument.

- Class type parameters appear to behave similarly, but differently, from primitive type parameters.
  - They appear to behave in a way similar to parameters in languages that have the ***pass-by-reference*** parameter passing mechanism.
  - However, they pass the address stored in the reference variable.

Class Parameters

- The value plugged into a class type parameter is a reference (memory address).
  - Therefore, the parameter becomes another name for the argument.
  - Any change made to the object referenced by the parameter will be made to the object referenced by the corresponding argument.
  - Any change made to the class type parameter itself (i.e., its address) will not change its corresponding argument (the reference or memory address).

Change Car Example

```
public class CarParameterTest {
    private static void installTurbocharger(int horsepower) {
        horsepower = horsepower + 20;
    }
    private static void changeCar1(Car car) {
        car = new Car("XYZ456", 2011, "Audi", "A8");
    }
    private static void changeCar2(Car car) {
        car.setStyle("Audi", "A8");
    }
    public static void main(String[] args) {
        Car car = new Car("ABC123", 1995, "Ford", "Mustang");
        installTurbocharger(car.getHorsepower());
        System.out.println(car); // output?
        changeCar1(car);
        System.out.println(car); // output?
        changeCar2(car);
        System.out.println(car); // output?
    }
}
```

Use of = and ==
with Variables of a Class Type

- The assignment operator (=) will produce two reference variables that name the same object.
- The test for equality (==) also behaves differently for class type variables.
  - The == operator only checks that two class type variables have the same memory address.
  - Unlike the `equals` method, it does not check that their instance variables have the same values.
  - Two objects in two different locations whose instance variables have exactly the same values would still test as being “not equal.”

The Constant null

- `null` is a special constant that may be assigned to a reference variable of any class type.

```
YourClass yourObject = null;
```

- Used to indicate that the variable has no "real value"
- Used in constructors to initialize class type instance variables when there is no obvious object to use
- `null` is not an object — it is, a kind of "placeholder" for a reference that does not name any memory location.
- Because it is like a memory address, use `==` or `!=` (instead of equals) to test if a reference variable contains `null`.

```
if(yourObject != null) {
    // we actually have an object instance
}
```

Anonymous Objects

- Recall, the `new` operator
  - Invokes a constructor which initializes an object, and
  - Returns a reference to the location in memory of the object created
- This reference can be assigned to a variable of the object's class type.
- Sometimes the object created is used as an argument to a method, and never used again.
  - In this case, the object need not be assigned to a variable, i.e., given a name.
- An object whose reference is not assigned to a variable is called an ***anonymous object***.

Anonymous Object Example

- An object whose reference is not assigned to a variable is called an ***anonymous object***.
- An anonymous Car object is used here as a parameter:

```
Car myCar = new Car("ABC123",2000,"Ford","Explorer");
if(myCar.equals(new Car("ABC123",2000,"Ford","Explorer"))) {
   System.out.println("Equal!");
}
```

- The above is equivalent to:

```
Car myCar = new Car("ABC123",2000,"Ford","Explorer");
Car temp = new Car("ABC123",2000,"Ford","Explorer");
if(myCar.equals(temp)) {
   System.out.println("Equal!");
}
```