# Classes and Objects: Object Creation and Constructors

CMSC 202H (Honors Section)

John Park

# Object Creation

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

```
Date myDate = new Date( );
```

- The expression

```
new Date( )
```

  invokes a special kind of method known as a ***constructor***.
- 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.
- It’s good programming practice to always implement a constructor with no parameters.

# The (Almost) Finished Date Class

```
public class Date
{
   private String month;
   private int day;            // 1 - 31
   private int year;           //4 digits

   // no-argument constructor
   // implementer chooses the default month, day, year
   public Date( )
   {
        month = "January";
        day = 1;
        year = 2007;
        // or better yet, call setDate(1, 1, 2007);
   }
   // alternative constructor
   public Date( int month, int day, int year )
   {
        this.month = monthString(month)
        this.day = day;
        this.year = year;
   }
```

(continued)

# Date Class (cont'd)

```
    // another alternative constructor
    // January 1 of the specified year
    public Date( int newYear )
    {
         this.month = monthString( 1 )
         this.day = 1;
         this.year = newYear;
    }

    // a constructor which makes a copy of an existing Date object
    // discussed in more detail later
    public Date( Date otherDate )
    {
             month = otherDate.month;
             day = otherDate.day;
             year = otherDate.year;
    }

    // remaining Date methods such as setDate, accessors, mutators
    // equals, toString, and stringMonth

} // end of Date class
```

# Using Date Constructors

```
public class DateDemo
{
   public static void main( String[ ] args)
   {
        Date birthday = new Date( 1, 23, 1982 );
        String s1 = birthday.toString( );     // January 23, 1982
        System.out.println( s1 );

        Date newYears = new Date( 2009 );
        String s2 = newYears.toString( );     // January 1, 2009
        System.out.println( s2 );

        Date holiday = new Date( birthday );
        String s3 = holiday.toString( );      // January 23, 1982
        System.out.println( s3 );

        Date defaultDate = new Date( );
        String s4 = defaultDate.toString( );  // January 1, 1000
        System.out.println( s4 );
   }
}
```

# this( ) Constructor

- When several alternative constructors are written for a class, we can reuse code by *calling one constructor from another*.

- The nested constructor is called as this( ).
- The call to this(…) *must* be the very first statement in the calling constructor
- You can execute other statements after the call to this()

# Constructor-chaining Designs

- Two design paradigms:
  - Complete base-level constructor:
    - Complex, full-functioned base-level constructor;

    Plus:
    - Calling constructors, that implement “defaults”
  - Primitive base-level constructor:
    - Minimum base constructor, that just fills in “un-defaultable” fields;

    Plus:
    - Calling constructors, that allow control of more fields

# Complete Base-level Constructor

```
// Most-general constructor, called by other constructors
public Date( int month, int day, int year )
{
      this.month = monthString(month)
      this.day = day;
      this.year = year;
}
// no-argument constructor
// implementer chooses the default month, day, year
public Date( )
{
      this( 1, 1, 2007 );
}

// alternative constructor: again, different defaults
// January 1 of the specified year
public Date( int newYear )
{
      this ( 1, 1, newYear );
}
```

# Primitive Base-level Constructor

```
// no-argument constructor
// Simplest version—just implements default values
public Date( )
{
      month = "January";
      day = 1;
      year = 2007;
}

// alternative constructor: allows more control
public Date( int newYear )
{
      this();
      year = newYear;
}

// most elaborate version (also using 'this' for fun)
public Date( int month, int day, int year )
{
      this(year);
      this.day = day;
      this.month = month;
}
```

# 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:

| Var | Value |
|---|---|
| x | 42 |
| y | 3.7 |

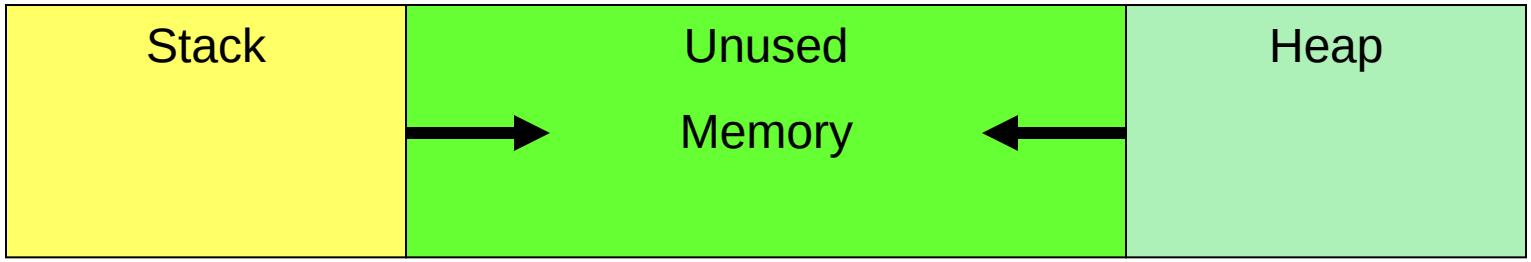
- The rest of memory is known as the ***heap*** and is used for ***dynamically allocated*** "stuff."

# 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 Dates:

```
Date d1, d2;
d1 = new Date(1, 1, 2000);
d2 = new Date(7, 4, 1776);
```

Where are these variables and objects located in memory?

Why do we care?

# Objects in Memory

The statement

```
Date d1, d2;
```

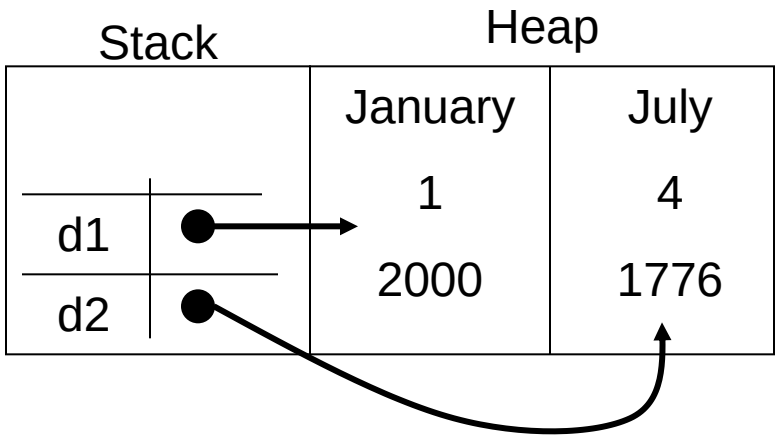
creates two local variables on the stack.

The statements

```
d1 = new Date(1, 1, 2000);
d2 = new Date(7, 4, 1776);
```

create objects on the heap. `d1` and `d2` contain the memory addresses of these objects giving us the picture of memory shown below.

`d1` and `d2` are called ***reference variables***. Reference variables which do not contain the memory address of any object contain the special value **null.**

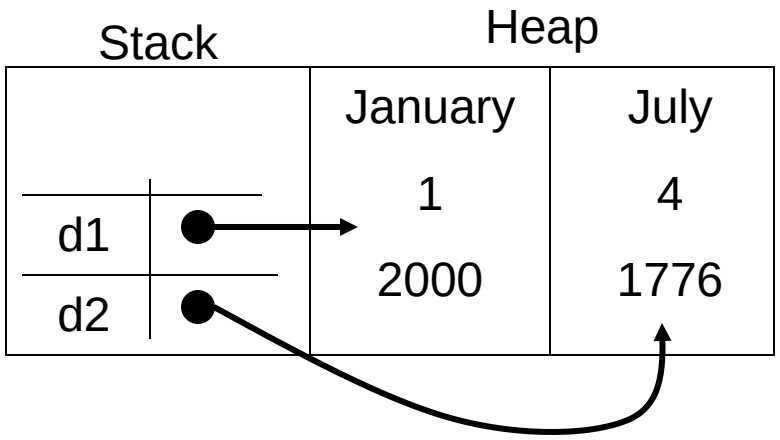


# Why We Care (1 of 4)

Given the previous code

```
Date d1, d2;
d1 = new Date(1, 1, 2000);
d2 = new Date(7, 4, 1776);
```

and corresponding picture of memory consider the expression `d1 == d2`




Recall that `d1` and `d2` contain the addresses of their respective Date objects. Since the Date objects have different addresses on the heap, `d1 == d2` is ***false***. The == operator determines if two reference variables refer to the same object.

So how do we compare Dates for equality?

Dates (and other objects) should implement a method named `equals`. To check if two Dates are the same, use the expression

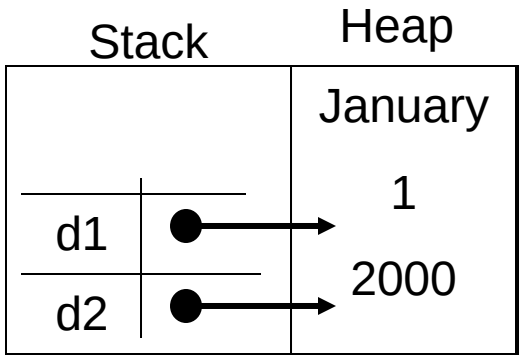
```
d1.equals( d2 );
```

.

# Why We Care (2 of 4)

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

```
Date d1 = new Date(1, 1, 2000);
Date d2 = d1;
```




Now d1 and d2 refer to the same Date object. This is known as ***aliasing***, is often unintentional, and can be dangerous. Why?

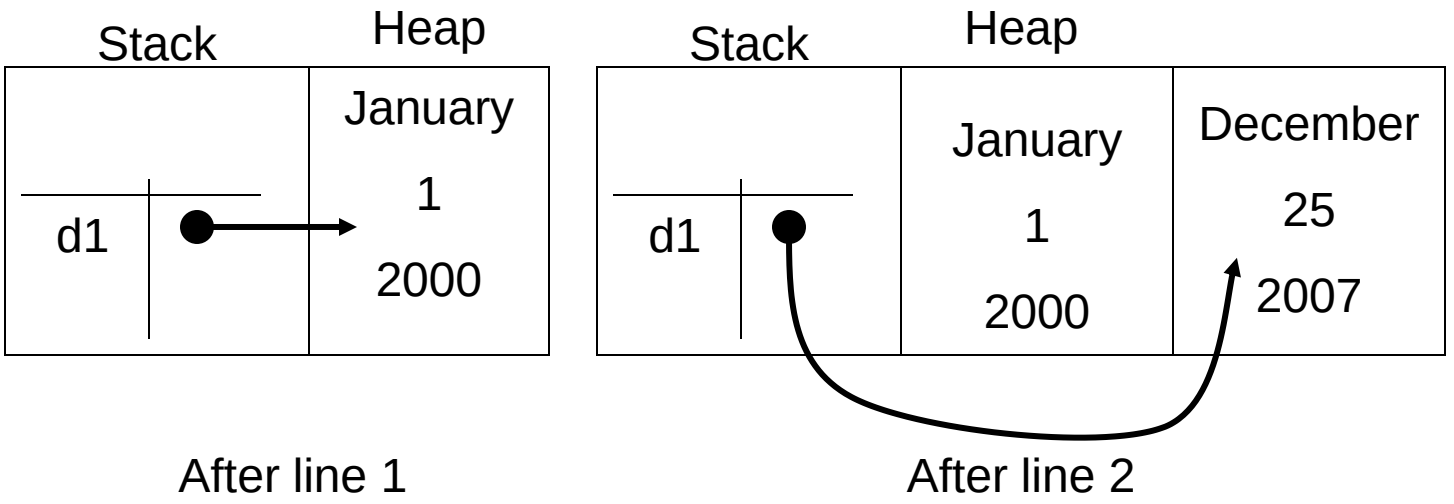
If your intent is for d2 to be a copy of d1, then the correct code is

```
Date d2 = new Date( d1 );
```

# Why We Care (3 of 4)

Consider this code and the changing picture of memory

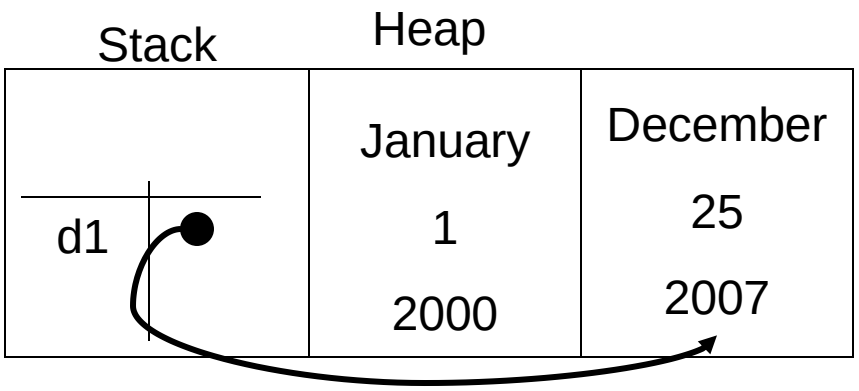
```
Date d1 = new Date(1, 1, 2000);     // line 1
d1 = new Date(12, 25, 2007);        // line 2
```



# Why We Care (4 of 4)

- Garbage collection

As the diagram shows, after line 2 is executed no variable refers to the Date object which contains “January”, 1, 2000




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 with a Class Base Type

- The base type of an array can be a class type as well as a primitive type.
- The statement

```
Date[] holidayList = new Date[20];
```

creates 20 indexed ***reference variables*** of type `Date`

  - It does not create 20 objects of the class `Date`.
  - 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 implemented as a location in computer 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.

(continued)

# 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 ***call-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 differently from primitive type parameters.
  - They appear to behave in a way similar to parameters in languages that have the ***call-by-reference*** parameter passing mechanism.

# 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).

# changeDay Example

```
public class DateParameterTest
{
   public static void changeDay (int day)
        { day = 1; }

   public static void changeDate1( Date aDate )
        { aDate = new Date( 1, 1, 2001); }

   public static void changeDate2( Date aDate )
        { aDate.setDate( 1, 1, 2001 ); }

   public static void main( String[ ] args )
   {
        Date birthday = new Date( 1, 23, 1982 );

        changeDay( birthday.getDay( ) );
        System.out.println(birthday.toString( ));    // output?

        changeDate1( birthday );
        System.out.println(birthday.toString( ));    // output?

        changeDate2( birthday );
        System.out.println(birthday.toString( ));    // 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)  . . .
```

# 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 Date object is used here as a parameter:

```
Date birthday = new Date( 1, 23, 1982 );
if (birthday.equals( new Date ( 1, 7, 2000 ) )
    System.out.println( "Equal!" );
```

- The above is equivalent to:

```
Date birthday = new Date( 1, 23, 1982 );
Date temp = new Date( 1, 7, 2000 );
if (birthday.equals( temp )
    System.out.println( "Equal!" );
```