Object Oriented Programming

Constructors, Statics, Wrappers & Packages

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 reuse code by calling one constructor from another.

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

Better Date Constructors

```
// no-argument constructor
// implementer chooses the default month, day, year
public Date( )
{
     this( 1, 1, 2007 );
}

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

// most general alternative constructor called by other
// constructors
public Date( int month, int day, int year )
{
     this.month = monthString(month)
     this.day = day;
     this.year = 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:

Var	Value
x	42
y	3.7

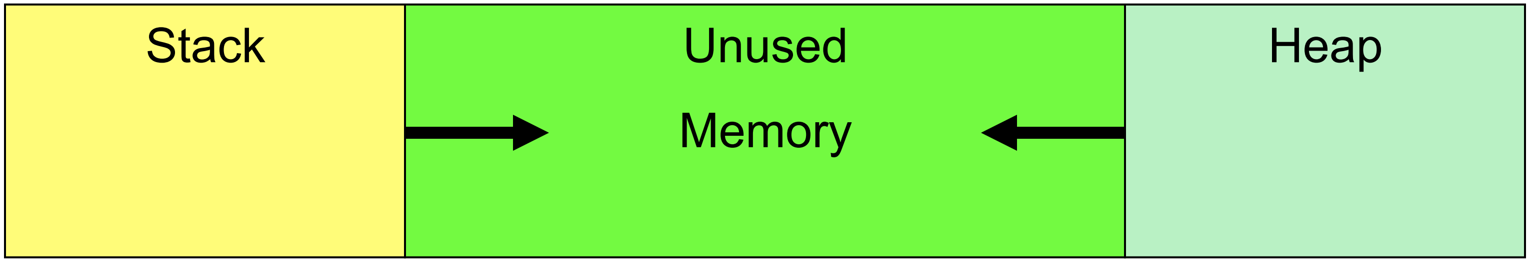
- The rest of memory is known as the ***heap*** and is used for ***dynamically allocated*** "stuff."

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;
```

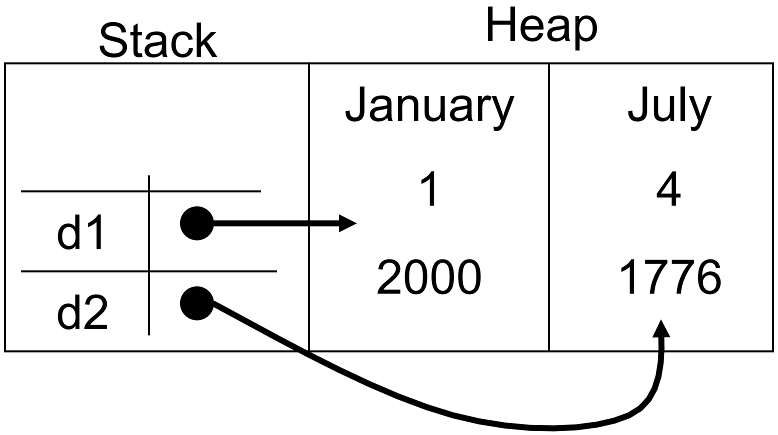
creates two local variables on the stack.

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.**

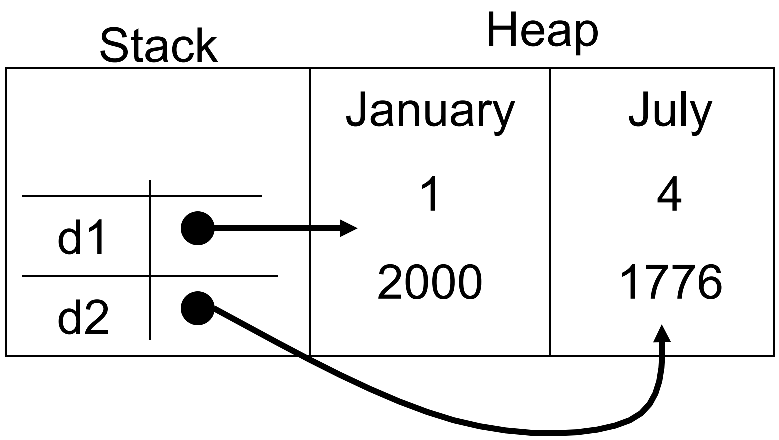


Why We Care (1 of 4)

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

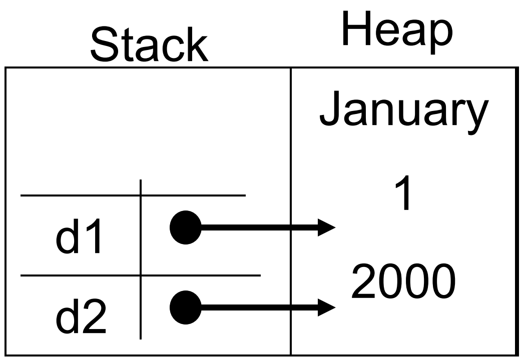
```
d1.equals( d2 );
```

.

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?

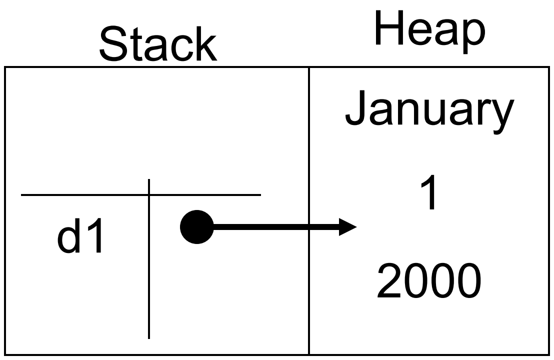
If your intent is for d2 to be a copy of d1, then the correct code is

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

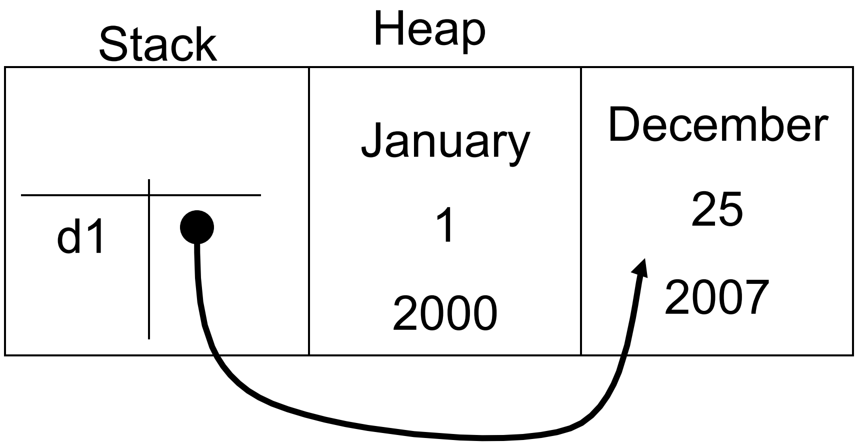
Why We Care (3 of 4)

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
```



After line 1

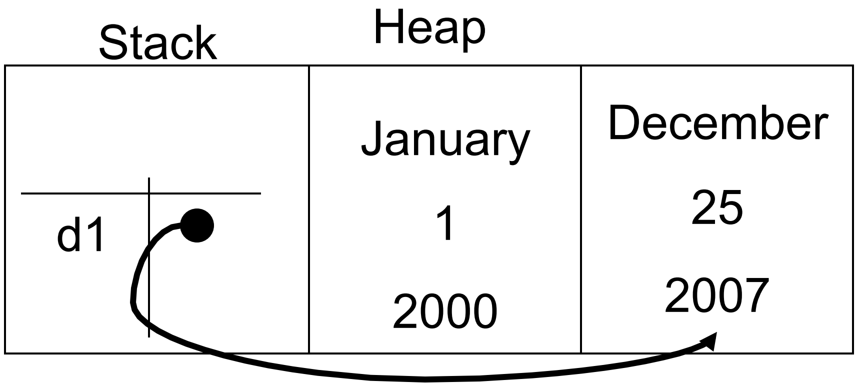


After line 2

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.

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.

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.

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!” );
```

Encapsulation

Encapsulation for Control

- We said we will use the term *encapsulation* in two different ways in this class (and in the text)
  - *Definition #1: "Inclusion" (“bundling"):*
    - *bundling of structure and function*
    - *Covered in lecture on “Object Design”*
  - **Definition #2: “Exclusion” (“access control”)**
    - **Strict, explicit control of how our objects can be used**
    - *This will be focus of this lecture*

Types of Programmers

- Class creators
  - those developing new classes
  - want to build classes that expose the minimum interface necessary for the ***client program*** and hide everything else
- Client programmers
  - those who use the classes (a term coined by Scott Meyer)
  - want to create applications by using a collection of interacting classes

OOP Techniques

- Class creators achieve their goal through ***encapsulation***.

  Encapsulation:

  - Combines data and operations into a single entity (a class)
  - Provides proper access control
  - Focuses on implementation
  - Achieved through ***information hiding*** (abstraction)

The Value of Encapsulation

- Client programmers do not need to know how the class is implemented, *only how to use it*.
- The information the client programmer needs to use the class is *kept to a minimum*.
- Class implementation may be changed *with no impact* on those who use the class.

Access Control

- Encapsulation is implemented using ***access control***.
  - Separates interface from implementation
  - Provides a boundary for the client programmer
- Visible parts of the class (the ***interface***)
  - can be used and/or changed by the client programmer.
- Hidden parts of the class (the ***implementation***)
  - Can be changed by the class creator without impacting any of the client programmer’s code
  - Can’t be corrupted by the client programmer

Access Control in Java

- ***Visibility modifiers*** provide access control to <u>instance variables</u> and <u>methods</u>.
  - ***public*** visibility - accessible by everyone, in particular the client programmer
    - A class' interface is defined by its public methods.
  - ***private*** visibility - accessible only by the methods within the class
  - Two others—***protected*** and [package]—later

Date2 Class

In this new date class, the instance variables have been labeled **private.**

```
public class Date2
{
  private String month;
  private int day;
  private int year;

  public void toString( )
  {
      return month + “ “ + day + “ “ + year;
  }
```

Any Date2 class method may use the class’ private instance variables.

```
  // setDate and monthString same as Date1 class
}
```

Access Control Example

Date1 class - **public** instance variables were used
Date2 class - **private** instance variables are now used

```
public class Date2Demo
{
  public static void main( String[ ] args )
  {
      Date2 myDate = new Date2( );

      myDate.month = “July”;   // compiler error
      myDate.day = 4;          // compiler error
      myDate.year = 1950;      // compiler error

      myDate.setDate( 7, 4, 1950 ); // OK - why?
      System.out.println( myDate.toString( ));
    }
}
```

Good Programming Practice

- Combine methods and data in a single class
- Label all instance variables as **private** for information hiding
  - The class has complete control over how/when/if the instance variables are changed
  - Instance variables primarily support class behavior
- Minimize the class' public interface

"Keep it secret, keep it safe."

Accessors & Mutator

- Class *behavior* may allow access to, or modification of, individual private instance variables.

- Accessor method
  - retrieves the value of a private instance variable
  - conventional to start the method name with **`get`**
- Mutator method
  - changes the value of a private instance variable
  - conventional to start the name of the method with **`set`**

- Gives the client program indirect access to the instance variables.

More Accessors and Mutators

Question: Doesn't the use of accessors and mutators defeat the purpose of making the instance variables `private`?

Answer: **No**

- The class implementer decides which instance variables will have accessors.
- Mutators can:
  - validate the new value of the instance variable, and
  - decide whether or not to actually make the requested change.

Date2 Accessor and Mutator

```
public class Date2
{
   private String month;
   private int day;        // 1 - 31
   private int year;       // 4-digit year

   // accessors return the value of private data
   public int getDay ( )
   { return day; }

   // mutators can validate the new value
   public boolean setYear( int newYear )
   {
        if ( 1000 <= newYear && newYear <= 9999 )
        {
            year = newYear;
            return true;
        }
        else // this is an invalid year
           return false;

   // rest of class definition follows
}
```

Accessor/Mutator Caution

- In general you should NOT provide accessors and mutators for all private instance variables.
  - Recall that the principle of encapsulation is best served with a *limited class interface*.
- Too many accessors and mutators lead to writing procedural code rather than OOP code. More on this later.

Classes as Structures

- There are two possible exceptions to the “make everything private” rule:
  - When the class is actually just a simple data structure
    - No hard consistency rules
    - No behaviors
    - Local use
  - When performance is critical
    - However, this tradeoff is often not worthwhile

Private Methods

- Methods may be private.
  - – Cannot be invoked by a client program
  - – Can only be called by other methods within the same class definition
  - – Most commonly used as “helper” methods to support top-down implementation of a public method

Private Method Example

```
public class Date2
{
   private String month;
   private int day;       // 1 - 31
   private int year;      // 4-digit year

   // mutators should validate the new value
   public boolean setYear( int newYear )
   {
         if ( yearIsValid( newYear ) )
         {
             year = newYear;
             return true;
         }
         else     // year is invalid
            return false;

   }
   // helper method - internal use only
   private boolean yearIsValid( int year )
   {
         return 1000 <= year && year <= 9999;
   }
}
```

Static and Final

The Problem of Words

"When I use a word," Humpty Dumpty said in rather a scornful tone, "it means just what I choose it to mean -- neither more nor less."

"The question is," said Alice, "whether you can make words mean so many different things."

"The question is," said Humpty Dumpty, "which is to be master - - that's all."

Lewis Carroll, *Through the Looking Glass*

- So, what do `static` (and `final`) mean in Java?
  - `public static final float PI = 3.14159;`
  - `public static int timesCreated;`
  - `public static void main(String[] args) {…}`
- …and *why* do they mean ***that***?!

History of `static`

- In C, originally needed a way to let a variable keep its value unchanged across calls, i.e., keep it "static"
- Extended scope to repurpose `static` keyword for file-scope global variables
- Java repurposed the word multiple times again, in an OOP context
- Humpty Dumpty would have loved `static`

What Does “static” Mean in Java?

- Instance variables, constants, and methods may all be labeled as `static`.

- In this context, static means that there is one copy of the variable, constant, or method that belongs to the class as a whole, and not to a particular instance.

- It is not necessary to instantiate an object to access a static variable, constant or method.

Static Variables

- A ***static variable*** belongs to the class as a whole, not just to one object.

- There is only one copy of a static variable per class.

- All objects of the class can read and change this static variable.

- A static variable is declared with the addition of the modifier `static`.

```
    static int myStaticVariable = 0;
```

Static Constants

- A ***static constant*** is used to symbolically represent a constant value.
- In Java, constants derive from regular variables, by "finalizing" them
  - The declaration for a static defined constant must include the modifier `final`, which indicates that its value cannot be changed.

```
public static final float PI = 3.142;
```

(The modifier `final` is also overloaded, and means other things in other contexts, as we shall see later.)

- Static constants belong to the class as a whole, not to each object, so there is only one copy of a static constant

- When referring to such a defined constant outside its class, use the name of its class in place of a calling object.

```
float radius = MyClass.PI * radius * radius;
```

Static Methods

So far,

- class methods required a calling object in order to be invoked.

```
Date birthday = new Date(1, 23, 1982);
String s = birthday.toString( );
```

- These are sometimes known as ***non-static methods***.

Static methods:

- still belong to a class, but need no calling object, and
- often provide some sort of utility function.

monthString Method

Recall the Date class private helper method monthString.

- Translates an integer month to a string
- Note that the monthString method
  - Does not call any other methods of the Date class, and
  - Does not use any instance variables (month, day, year) from the Date class.

- This method can be made available to users of the Date class without requiring them to create a Date object.

```
public static String monthString( int monthNumber ) {
     switch ( monthNumber )  {
             case 1:  return "January";
             case 2:  return "February";
             case 3:  return "March";
             case 4:  return "April";
             case 5:  return "May";
             case 6:  return "June";
             case 7:  return "July";
             case 8:  return "August";
             case 9:  return "September";
             case 10: return "October";
             case 11: return "November";
             case 12: return "December";
             default: return “????”;
     }
}
```

It is now a public static method.

monthString Demo

- Code outside of the Date class can now use the monthString method without creating a Date object.
- Prefix the method name with the name of the class instead of an object.

```
class MonthStringDemo
{
   public static void main( String [ ] args )
   {
         String month = Date.monthString( 6 );
         System.out.println( month );
   }
}
```

Date is a class name, not an object name.

monthString is the name of a static method

Rules for Static Methods

- Static methods have no calling/host object (they have no `this`).

- Therefore, static methods cannot:
  - Refer to any instance variables of the class
  - Invoke any method that has an implicit or explicit `this` for a calling object

- Static methods may invoke other static methods or refer to static variables and constants.

- A class definition may contain both static methods and non-static methods.

Static F^o to C^o Convert Example

```
public class FtoC
{
   public static double convert( double degreesF )
         { return 5.0 / 9.0 * (degreesF - 32 ); }
}
```

```
public class F2CDemo
{
   public static void main( String[ ] args )
   {
         double degreesF = 100;

         // Since convert is static, no object is needed
         // The class name is used when convert is called

         double degreesC = FtoC.convert( degreesF );
         System.out.println( degreesC );
   }
}
```

main is a Static Method

Note that the method header for main( ) is

```
public static void main(String [] args)
```

Being static has two effects:

- main can be executed without an object.
- “Helper” methods called by main must also be static.

Any Class Can Have a main( )

- Every class can have a public static method name main( ).
- Java will execute main in whichever class is specified on the command line.

```
java <className>
```

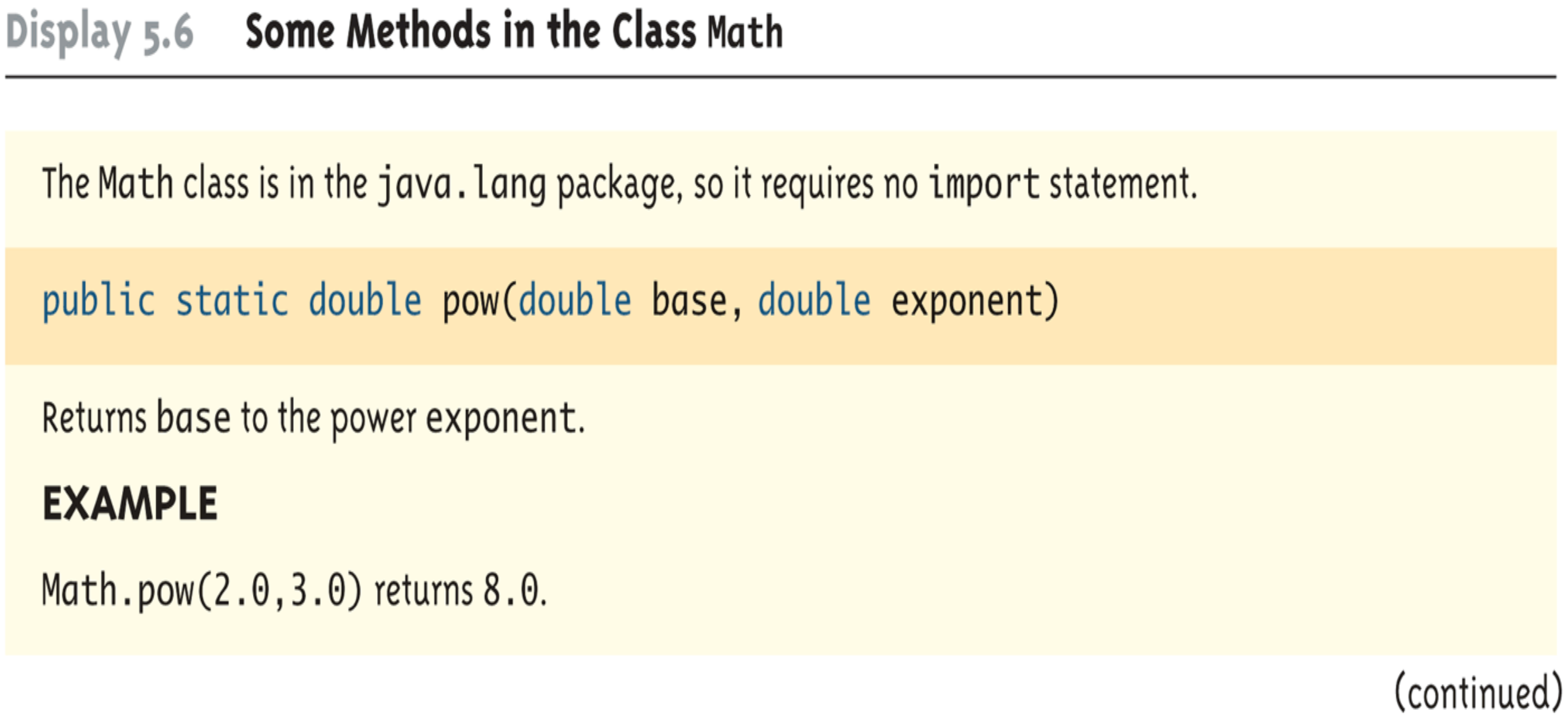
- A convenient way to write test code for your class.

The **`Math`** Class

- The **`Math`** class provides a number of standard mathematical methods.
  - Found in the **`java.lang`** package, so it does not require an **`import`** statement
  - All of its methods and data are static.
    - They are invoked with the class name **`Math`** instead of a calling object.
  - The **`Math`** class has two predefined constants, **`E`** (*e*, the base of the natural logarithm system) and **`PI`** (π, 3.1415 . . .).

```
area = Math.PI * radius * radius;
```

Some Methods in the Class **`Math`** (Part 1 of 5)



Display 5.6 **Some Methods in the Class `Math`**

The `Math` class is in the `java.lang` package, so it requires no `import` statement.

```
public static double pow(double base, double exponent)
```

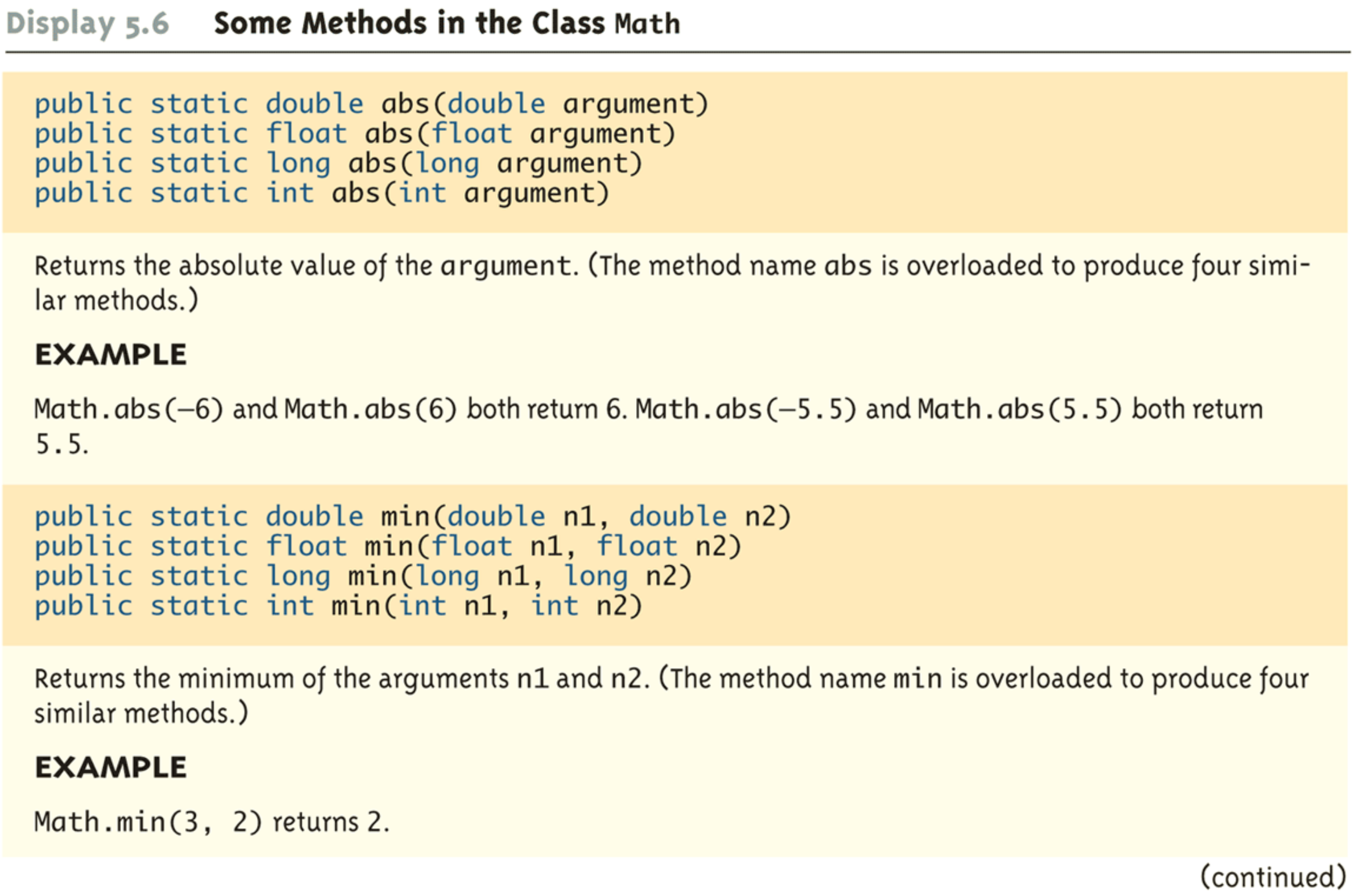
Returns base to the power exponent.

EXAMPLE

`Math.pow(2.0,3.0)` returns `8.0`.

(continued)

Some Methods in the Class **Math** (Part 2 of 5)



Display 5.6 **Some Methods in the Class Math**

```
public static double abs(double argument)
public static float abs(float argument)
public static long abs(long argument)
public static int abs(int argument)
```

Returns the absolute value of the argument. (The method name abs is overloaded to produce four similar methods.)

EXAMPLE

Math.abs(–6) and Math.abs(6) both return 6. Math.abs(–5.5) and Math.abs(5.5) both return 5.5.

```
public static double min(double n1, double n2)
public static float min(float n1, float n2)
public static long min(long n1, long n2)
public static int min(int n1, int n2)
```

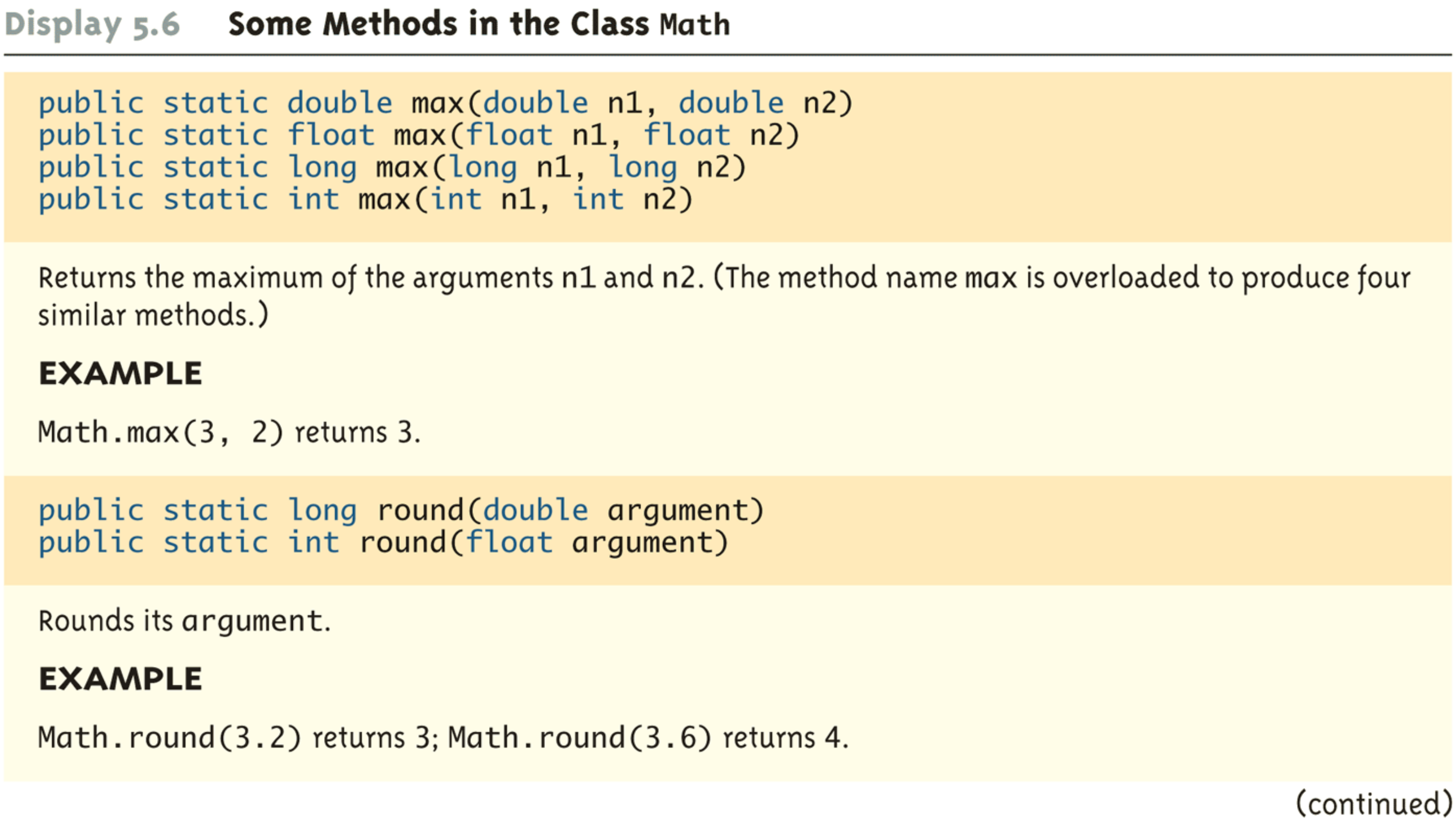
Returns the minimum of the arguments n1 and n2. (The method name min is overloaded to produce four similar methods.)

EXAMPLE

Math.min(3, 2) returns 2.

(continued)

Some Methods in the Class **`Math`** (Part 3 of 5)



Display 5.6 **Some Methods in the Class `Math`**

```
public static double max(double n1, double n2)
public static float max(float n1, float n2)
public static long max(long n1, long n2)
public static int max(int n1, int n2)
```

Returns the maximum of the arguments `n1` and `n2`. (The method name `max` is overloaded to produce four similar methods.)

EXAMPLE

`Math.max(3, 2)` returns 3.

```
public static long round(double argument)
public static int round(float argument)
```

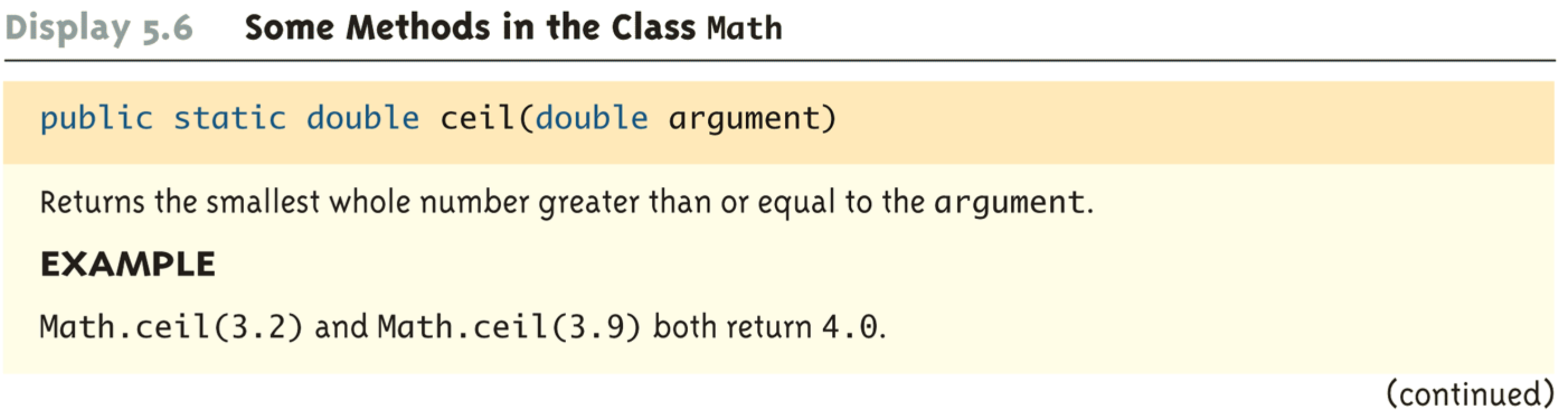
Rounds its argument.

EXAMPLE

`Math.round(3.2)` returns 3; `Math.round(3.6)` returns 4.

(continued)

Some Methods in the Class **`Math`** (Part 4 of 5)



Display 5.6 Some Methods in the Class `Math`

```
public static double ceil(double argument)
```

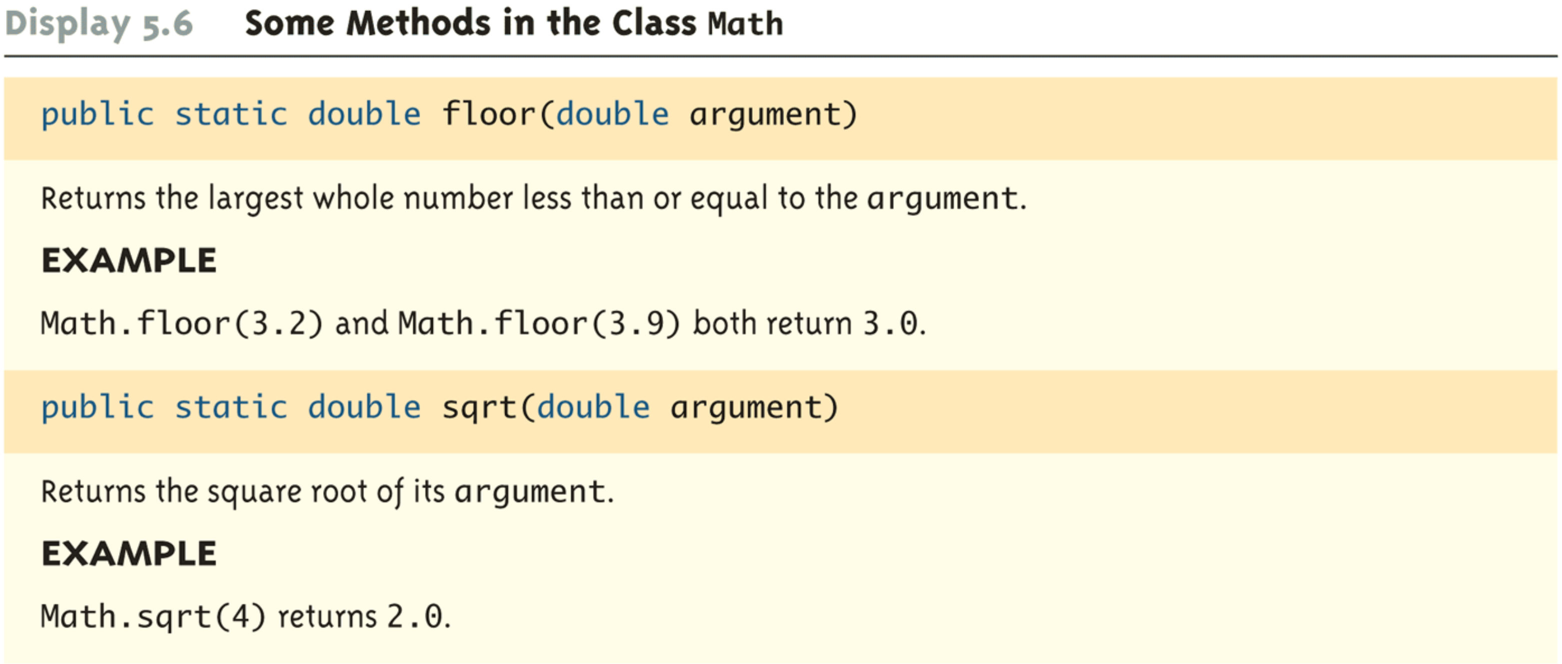
Returns the smallest whole number greater than or equal to the `argument`.

EXAMPLE

`Math.ceil(3.2)` and `Math.ceil(3.9)` both return `4.0`.

(continued)

Some Methods in the Class **`Math`** (Part 5 of 5)



Display 5.6 **Some Methods in the Class `Math`**

```
public static double floor(double argument)
```

Returns the largest whole number less than or equal to the `argument`.

EXAMPLE

`Math.floor(3.2)` and `Math.floor(3.9)` both return `3.0`.

```
public static double sqrt(double argument)
```

Returns the square root of its `argument`.

EXAMPLE

`Math.sqrt(4)` returns `2.0`.

Static Review

- Given the skeleton class definition below

```
public class C {
  public int a = 0;
  public static int b = 1;

  public void f() { …}
  public static void g() {…}
}
```

- Can body of f() refer to a?
- Can body of f() refer to b?
- Can body of g() refer to a?
- Can body of g() refer to b?
- Can f() call g()?
- Can g() call f()?

For each, explain why or why not.

Wrapper Classes

- ***Wrapper classes***
  - Provide a class type corresponding to each of the primitive types
  - Makes it possible to have class types that behave somewhat like primitive types
  - The wrapper classes for the primitive types:

`byte`, `short`, `int`, `long`, `float`, `double`, and `char`

are (in order)

`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, and `Character`

  - Wrapper classes also contain useful
    - predefined constants
    - static methods

Constants and Static Methods in Wrapper Classes

- Wrapper classes include constants that provide the largest and smallest values for any of the primitive number types.
  - **`Integer.MAX_VALUE`**, **`Integer.MIN_VALUE`**, **`Double.MAX_VALUE`**, **`Double.MIN_VALUE`**, etc.

- The **`Boolean`** class has names for two constants of type **`Boolean.`**
  - **`Boolean.TRUE`** corresponds to **`true`**
  - **`Boolean.FALSE`** corresponds to **`false`**

of the primitive type **`boolean.`**

Constants and Static Methods in Wrapper Classes

- Some static methods convert a correctly formed string representation of a number to the number of a given type.
  - The methods `Integer.parseInt()`, `Long.parseLong()`, `Float.parseFloat()`, and `Double.parseDouble()`

    do this for the primitive types (in order) `int`, `long`, `float`, and `double`.
- Static methods convert from a numeric value to a string representation of the value.
  - For example, the expression

    ```
    Double.toString(123.99);
    ```

    returns the string value `"123.99"`
- The `Character` class contains a number of static methods that are useful for string processing.

Wrappers and Command Line Arguments

- Command line arguments are passed to main via its parameter conventionally named args.

```
public static void main (String[ ] args)
```

- For example, if we execute our program as

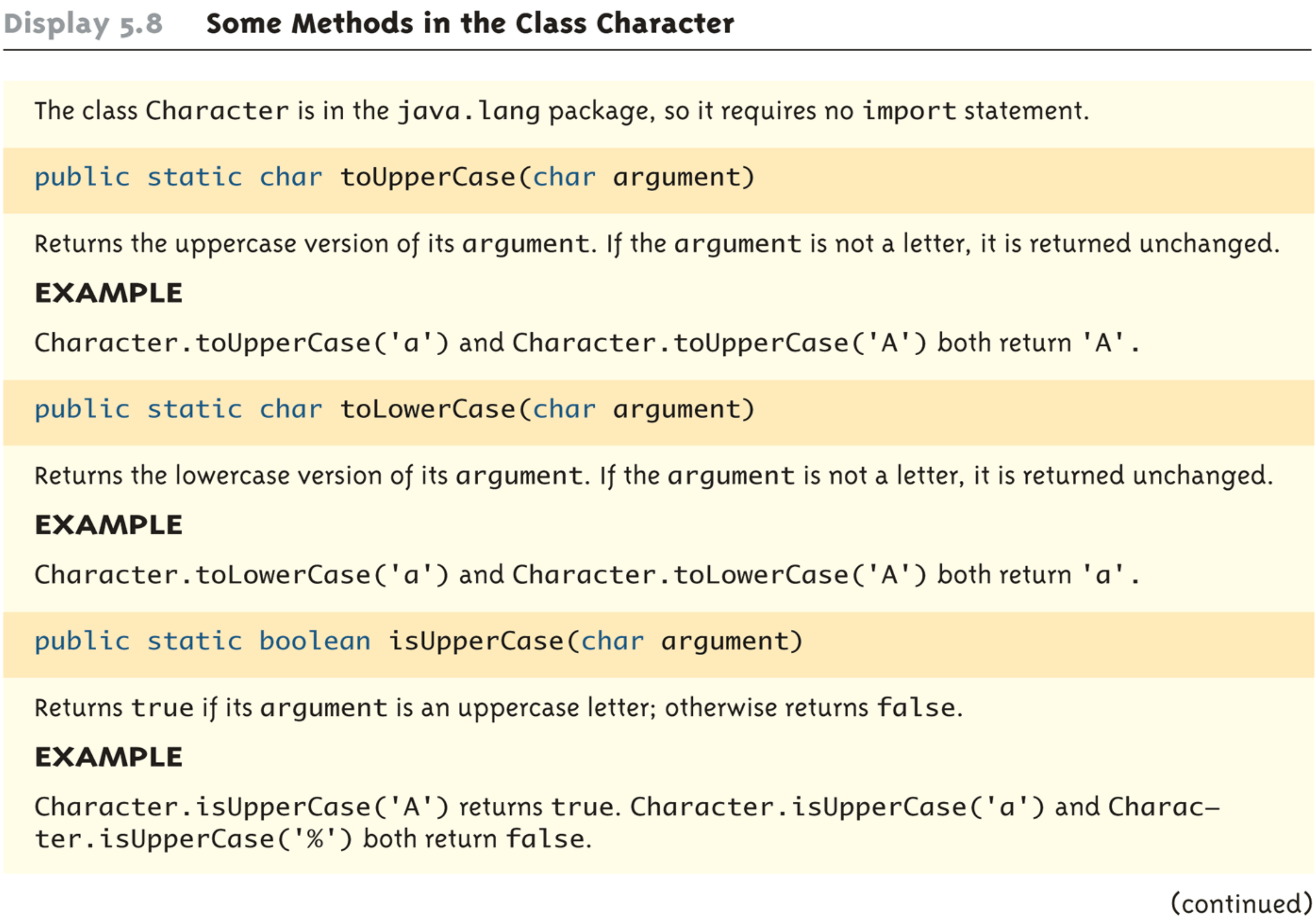
```
java proj1.Project1 Bob 42
```

then args[0] = “Bob” and args[1] = “42”.

- We can use the static method **Integer.parseInt( )** to change the argument “42” to an integer variable via

```
int age = Integer.parseInt( args[ 1 ] );
```

Methods in the Class `Character` (1 of 3)



Display 5.8 Some Methods in the Class Character

The class `Character` is in the `java.lang` package, so it requires no `import` statement.

`public static char toUpperCase(char argument)`

Returns the uppercase version of its `argument`. If the `argument` is not a letter, it is returned unchanged.

EXAMPLE

`Character.toUpperCase('a')` and `Character.toUpperCase('A')` both return `'A'`.

`public static char toLowerCase(char argument)`

Returns the lowercase version of its `argument`. If the `argument` is not a letter, it is returned unchanged.

EXAMPLE

`Character.toLowerCase('a')` and `Character.toLowerCase('A')` both return `'a'`.

`public static boolean isUpperCase(char argument)`

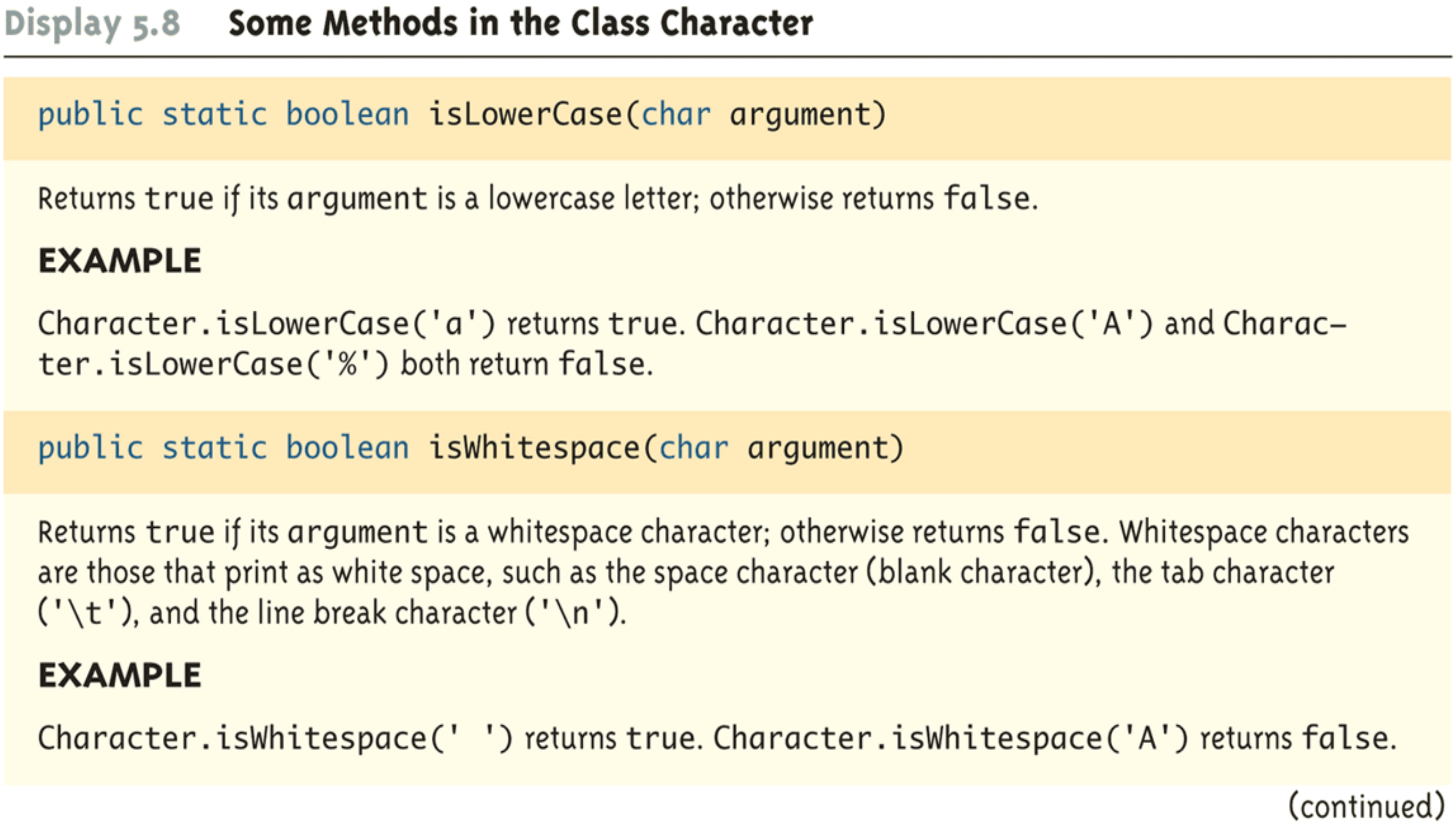
Returns `true` if its `argument` is an uppercase letter; otherwise returns `false`.

EXAMPLE

`Character.isUpperCase('A')` returns `true`. `Character.isUpperCase('a')` and `Character.isUpperCase('%')` both return `false`.

(continued)

Methods in the Class **Character** (2 of 3)



Display 5.8 Some Methods in the Class Character

`public static boolean isLowerCase(char argument)`

Returns `true` if its `argument` is a lowercase letter; otherwise returns `false`.

EXAMPLE

`Character.isLowerCase('a')` returns `true`. `Character.isLowerCase('A')` and `Character.isLowerCase('%')` both return `false`.

`public static boolean isWhitespace(char argument)`

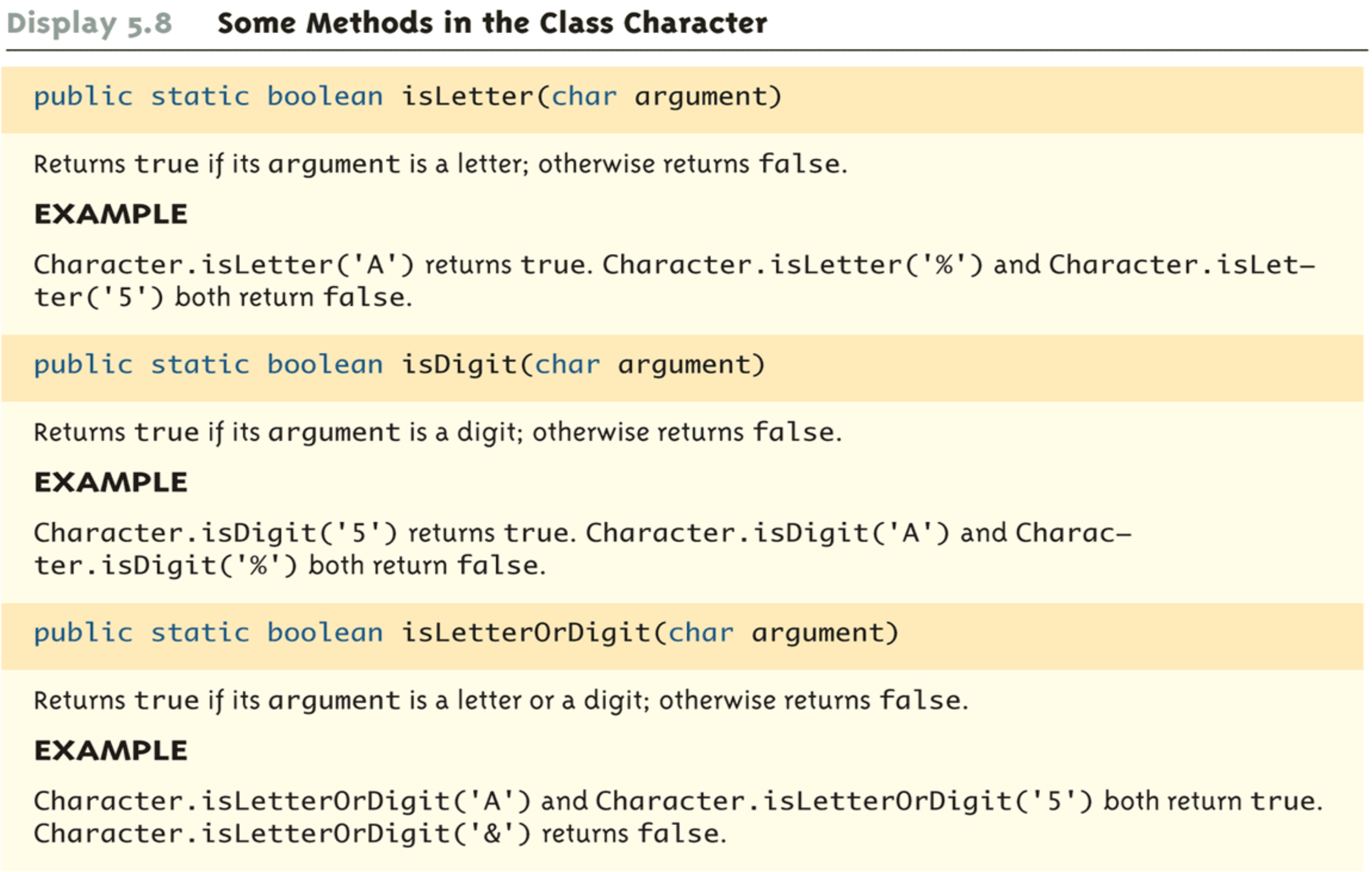
Returns `true` if its `argument` is a whitespace character; otherwise returns `false`. Whitespace characters are those that print as white space, such as the space character (blank character), the tab character (`'\t'`), and the line break character (`'\n'`).

EXAMPLE

`Character.isWhitespace(' ')` returns `true`. `Character.isWhitespace('A')` returns `false`.

(continued)

Methods in the Class **Character** (3 of 3)



Display 5.8 **Some Methods in the Class Character**

```
public static boolean isLetter(char argument)
```

Returns true if its argument is a letter; otherwise returns false.

EXAMPLE

Character.isLetter('A') returns true. Character.isLetter('%') and Character.isLetter('5') both return false.

```
public static boolean isDigit(char argument)
```

Returns true if its argument is a digit; otherwise returns false.

EXAMPLE

Character.isDigit('5') returns true. Character.isDigit('A') and Character.isDigit('%') both return false.

```
public static boolean isLetterOrDigit(char argument)
```

Returns true if its argument is a letter or a digit; otherwise returns false.

EXAMPLE

Character.isLetterOrDigit('A') and Character.isLetterOrDigit('5') both return true. Character.isLetterOrDigit('&') returns false.

Boxing

- ***Boxing***: The process of converting from a value of a primitive type to an object of its wrapper class.
  - Create an object of the corresponding wrapper class using the primitive value as an argument
  - The new object will contain an instance variable that stores a copy of the primitive value.

```
Integer integerObject = new Integer(42);
```

  - Unlike most other classes, a wrapper class does not have a no-argument constructor.
  - The value inside a Wrapper class is ***immutable***.

Unboxing

- ***Unboxing***: The process of converting from an object of a wrapper class to the corresponding value of a primitive type.
  - The methods for converting an object from the wrapper classes

    `Byte`, **`Short`**, **`Integer`**, **`Long`**, **`Float`**, **`Double`**, and **`Character`**

    to their corresponding primitive type are (in order)

    `byteValue`, **`shortValue`**, **`intValue`**, **`longValue`**, **`floatValue`**, **`doubleValue`**, and **`charValue`**.

  - None of these methods take an argument.

```
int i = integerObject.intValue();
```

Automatic Boxing and Unboxing

Starting with version 5.0, Java can automatically do boxing and unboxing for you.

- Boxing:

```
Integer integerObject = 42;
```

rather than:

```
Integer integerObject = new Integer(42);
```

- Unboxing:

```
int i = integerObject;
```

rather than:

```
int i = integerObject.intValue();
```

Packages

- Java allows you to partition your classes into sets and subsets, called *packages.*
- You place your class into a package with the directive:

```
package myPackage;
```

- If the "package" directive is missing, the class is placed into the *unnamed package*
- A Java package is similar to a "namespace": it implicitly prepends a prefix of your choice to all classes you define.

Packages

- You can refer to all objects via its fully-qualified name, e.g.:

```
myPackage.MyClass foo = new myPackage.MyClass();
```

- Within a class definition, class references without explicit package name prefixes refer to other classes in your package
  - This is modified by importing other packages
- In addition to its use for namespaces, packages affect the function of some *visibility modifiers* (later)

Importing Packages

- Import single *class* by using:

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

- Or, import many classes, with wildcard:

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

  - Cannot "`import java.*.*;`"
  - Importing is not recursive (e.g. java.* != java.util.*)
  - Importing singly is preferred (why?)
- java.lang.* is already implicitly imported
- However, all other java.*… must be explicitly imported

Package Naming Conventions

- Initially, beginners use the *unnamed package*
- For simple, standalone applications, use simple one-token package names, e.g.: “proj1” (note lowercase)
- For packages to be deployed outside the organization, use inverse-domain-address-like notation, e.g.:

  edu.brynmawr.cs.cmsc206.utilityPackage

Packages: Example

```
package proj3;
import java.util.Random;
public class MyClass {
     // Stuff inside this class definition
     public static int someMethod() {
             Random rand = new Random();
             …
     }
}
----------------------------------------------------------

// No “package” directive, so in unnamed package
// No “import” directive, so all class names must be full
public class MyOtherClass {
     // Stuff inside this class definition
     public static int someMethod() {
             proj3.MyClass myClassInst = new proj3.MyClass();
             java.util.Random rand = new java.util.Random();
             …
     }
}
```