ECE25100:
OBJECT ORIENTED PROGRAMMING

Note 4 _ Defining Class and Objects

Instructor: Xiaoli Yang

Class and instance

- By creating a class and writing its code, we are NOT actually creating any objects. We are merely defining them.
- A class is simply a category, not an object!
- We need to create (or instantiate) objects before we use them. A created object of class X is called an instance of class X.

Class and instance(Cont.)

- Logically, an instance:
  - is a member of a specific class.
    - e.g., Car instances are members of the Car class.
  - is a bundle of data that describes the state of an individual member of a class.
  - has a state which differs from other instances of that class.
    - e.g., all cars have common state of color, but some cars are red, some blue, some silver etc...
- So, objects belonging to a particular class will have the same defined state, but the values will be different.

Class and instance(Cont.)

- We can make a new instance of a class by saying:

  new <classname>( )

- Objects that have been created and are no longer being "pointed to" by anyone are "garbage collected". The garbage collector :
  - is a low priority process running in the Java Virtual Machine
  - is used to free up memory for unused objects
  - is necessary to release resources
  - runs automatically, programmer needs not do anything
  - can be forced to run by using System.gc()

Variables and Behaviors of a Class (review _ note 1)

- State is written usually right at the top of the class definition. Each part of the state (each attribute) has a type as well as its own name.
- Each behavior is written one after another just beneath the state.

```
public class ClassName {
  // These represent the state of the object
   type1    name1;
   type2    name2;
   ...
  // These represent the behavior of the object
   type1 nameOfBehaviour1() {
      // code defining behavior 1
   }
   type2 nameOfBehaviour2() {
      // code defining behavior 2
   }
   ...
}
```

1. State

Example:

Consider a Car object. What state do we want for a Car ?

It depends on the application! Here are some possible applications in which a Car object may be used:

- repair shop
  - make, model, year, engine size, spark plug type, air/oil filter types, air hose diameter, repair history, owner etc...
- rental agency
  - sedan or coupe, make, model, license plate, price per hour, mileage, repair history, etc...
- insurance company
  - year, make, model, owner, insurance type (fire/theft/collision/liability), color, license plate, etc...

- we represent the object's state through the use of instance variables (or fields).

Instance Variables

- An instance variable (or field):
  - is a variable which has a type (int, float, char, String, Color, etc.) and a modifier (public, private, etc.)
  - stores a common piece of information that all instances require
  - may have a value that differs from other instances of this same class
- Examples
  - Consider a class that represents a circle. what state (i.e., information) do all Circles have ?
  - what state (i.e., information) do all Rectangles have?

Instance Variables (Cont.)

- Examples

We can define a class as having a radius as its state like this:

```
public class Circle{
     float radius;
 }
```

We can define a Rectangle that stores a width and height as doubles:

```
public class Rectangle{
    double width;
    double height;
 }
```

Class Variables

- Sometimes there are common values for state pertaining to all instances of a class:
  - number of wheels for cars = 4
  - interest rate for all banks = 0.1 %
  - diameter of all CDs = 4.75 inches
  - pi = 3.14159265
- These values do not change between instances and are therefore called static variables (or class variables). A class variable (or static variable):
  - is a variable that is common to all members of a class.
  - stores information pertaining to the entire class.
  - must be declared as static

Class Variables

- Static variables are also used to store a commonly accessed value such as the last bank account number that was given out to the last customer served from a service center. In a sense, these are acting as global counters. Whenever a new account is created or a new customer served, the value is incremented:
  - static int LAST_ACCOUNT_NUMBER = 1;
  - static int LAST_NUMBER_SERVED = 66;

  More examples:
  - static double AVERAGE_HEIGHT = 5.5;
  - static Color MOST_COMMON_COLOR = new Color(132, 220, 86);
  - static String PAY_DAY = "Friday";

Class Constant

- If the value will never change within the program, we can add the keyword final so as to make it a constant. This is known as a static constant or class constant which will be shared (i.e., accessible) with everyone. Here are some examples:
  - static final double PI = 3.14159265;

    // Although this is available as Math.PI
  - static final Color R = new Color(255, 0, 0);

    // Although this is available as Color.RED
  - static final String WEEKEND = "Saturday";
- We can access the class variables through either the class itself or through an instance:

  <Classname or anInstance>.<classVariable or classConstant>

2. Creating Behaviors

- The interesting part of OO-Programming is getting the objects to interact together. This is obvious when we look at real world examples:
  - A BankAccount in which no money is deposited or withdrawn is not useful.
  - A CD without a CD Player is useless.
- Behavior represents:
  - the things you can do with an object (i.e., a command)
  - information you can ask for from an object (i.e., a question)

Defining Class Behaviors (Cont.)

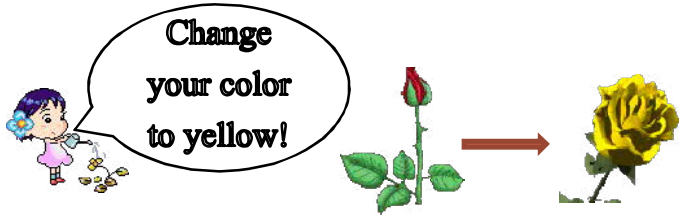
- Since you are defining/inventing your own objects, you get to decide what the object should be able to do, and get these objects to somehow "communicate" with each other.
- To help you understand object behavior, you should try to think of objects as being "living" entities. When we want to "talk to" or "manipulate" an object, we must send it a message.
- A message:
  - is a set of one or more words (joined together as one) that is sent to an object.
  - is part of the "vocabulary" that an object understands.
- may have additional information (parameters) which are required by the object.
- You can send messages to objects, and they respond to you.

Defining Class Behaviors (Cont.)

- You can send messages to objects, and they respond to you:

Typical java code to do this:

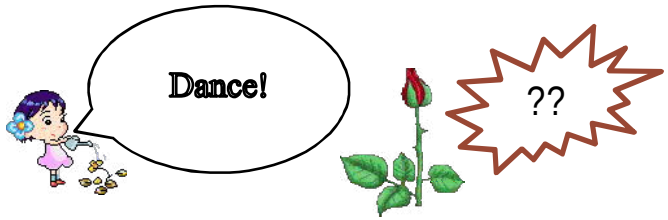
Color = aFlower.getColor();



Typical java code to do this:

aFlower.setColor("yellow");

- Objects only respond if they understand what you say:



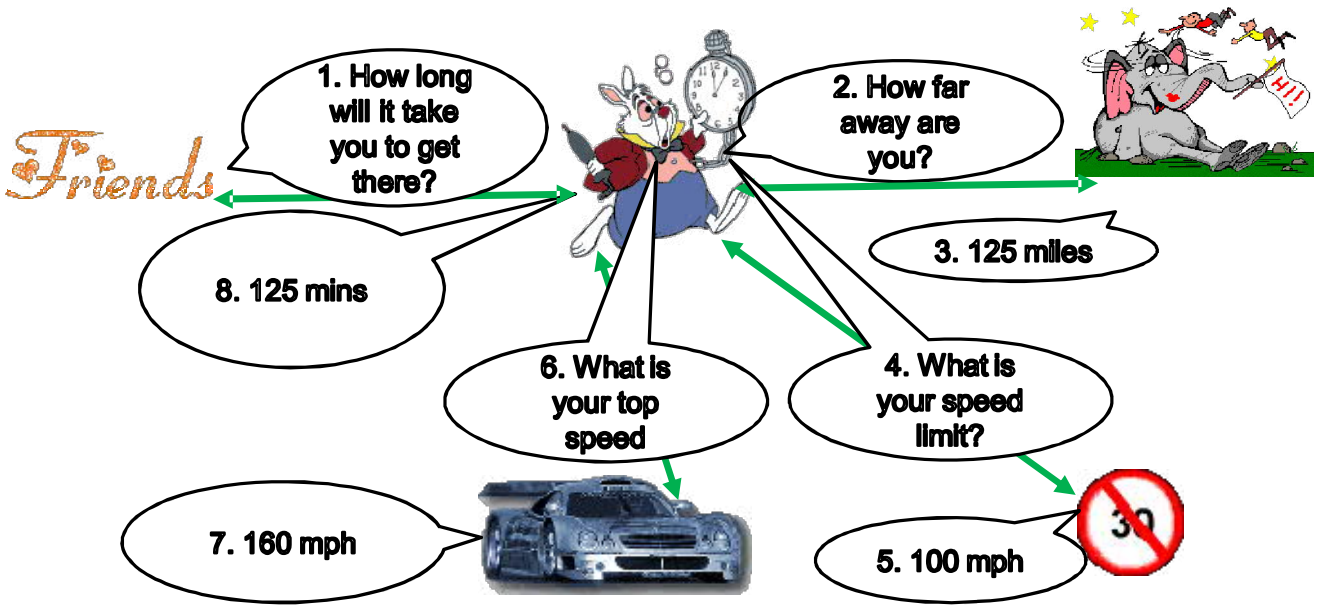
Typical java code to do this:

aFlower.dance();
//will be a compile error

http://www.animatedimages.org/

Defining Class Behaviors (Cont.)

- Thus, by defining behavior, we simply add to messages that the object understands.
  Objects communicate by sending messages back and forth to each other:



http://www.animatedimages.org/

Defining Class Behaviors (Cont.)

To define a particular behavior for an object, we must write a method (much like writing a library of functions and procedures in C).

A method :

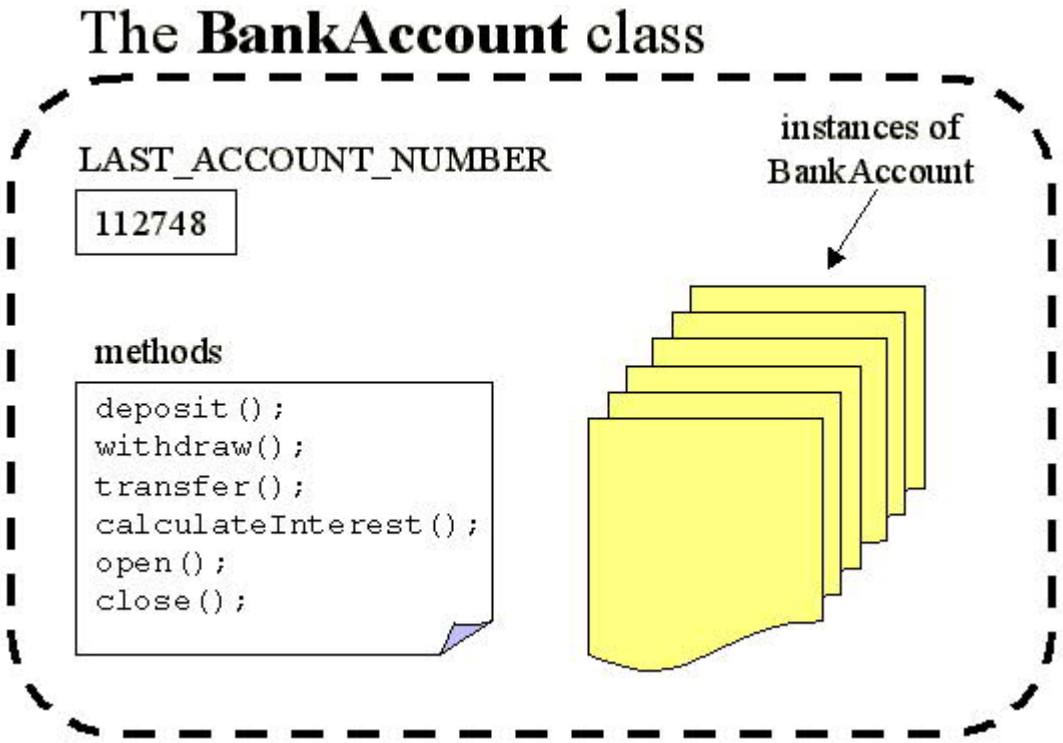
- is the code (expressions) that defines what happens when a message is sent to an object.
- may require zero or more parameters (i.e., pieces of data):
  - Parameters may be primitives or other objects
  - Primitives are "passed-by-value" (the actual value is "copied" and passed with the message)
  - Objects are "passed-by-reference" (a pointer to the object is passed with the message)
- may be either a class method or an instance method.

Defining Class Behaviors (Cont.)

- Methods are typically used to do one or more of these things:
  - get information from the object it is sent to
  - change the object in some way
  - compute or do something with the object
  - obtain some results.
- Methods are defined ONCE for the class and so they become a part of the class definition.

Defining Class Behaviors (Cont.)

- For example, a BankAccount class may define methods



Introducing Methods

- A method is a collection of statements that are grouped together to perform an operation.

- So, sending a message to an object really means calling one of its methods. We send messages to objects (i.e., instances) in JAVA by using the dot operator:

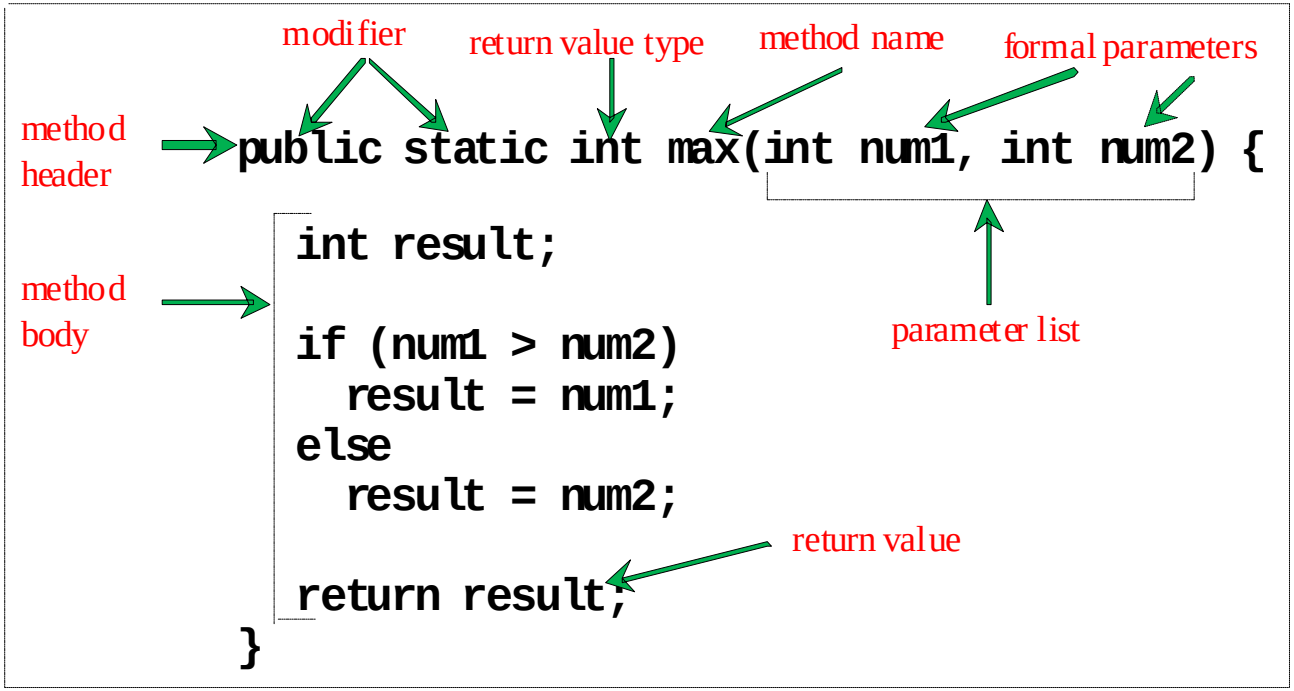
  <anObject>.<methodName>();

  - This means: "send the message called <methodName> to <anObject>". We can also supply parameter values within the round brackets ( ).

- if <anObject> here is null, we get a NullPointerException error.

Calling Methods, cont.

Define a method

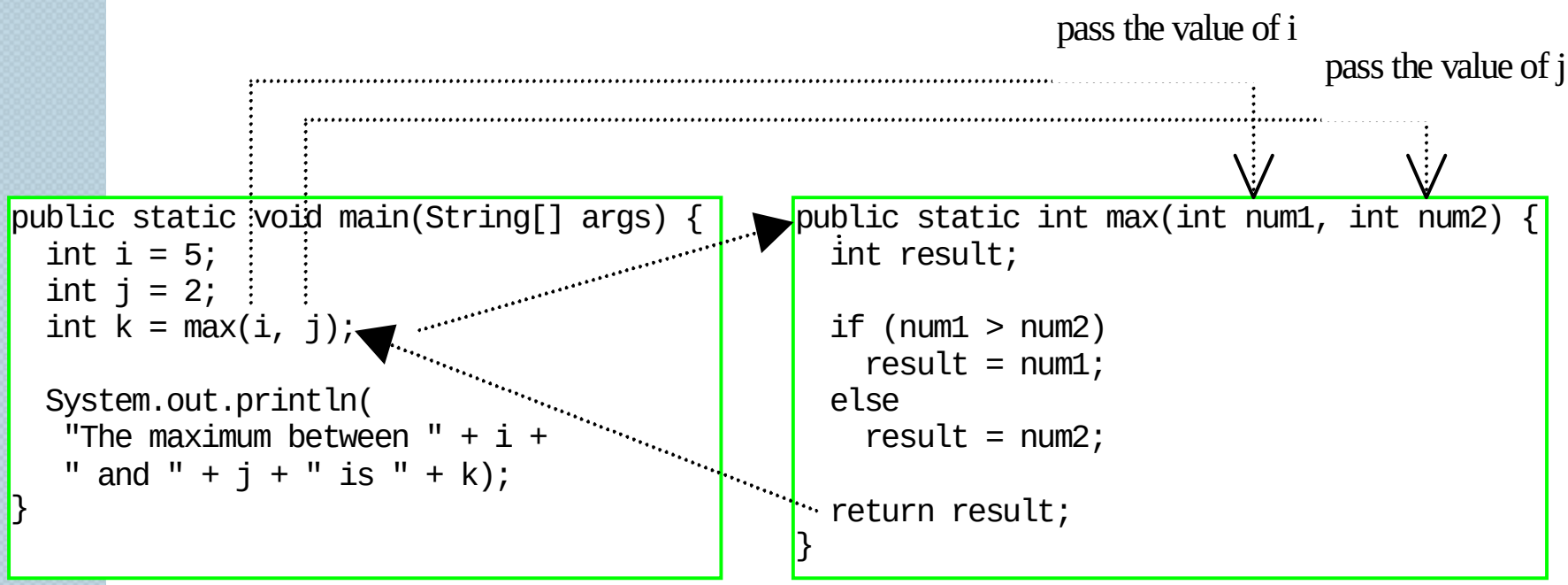


Introducing Methods (cont.)

- Method signature is the combination of the method name and the parameter list.
- The variables defined in the method header are known as formal parameters.
- When a method is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument.
- A method may return a value. The return Value Type is the data type of the value the method returns. It represents the "answer" to what is being asked of the object.
- If the method does not return a value (there is no answer), the return Value Type is the keyword void. For example, the return ValueType in the main method is void.

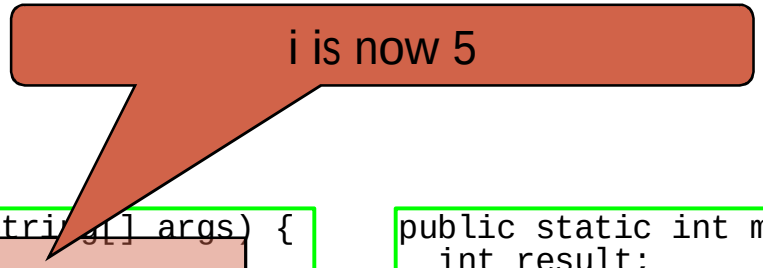
animation

Calling Methods, cont.



animation

Trace Method Invocation




```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

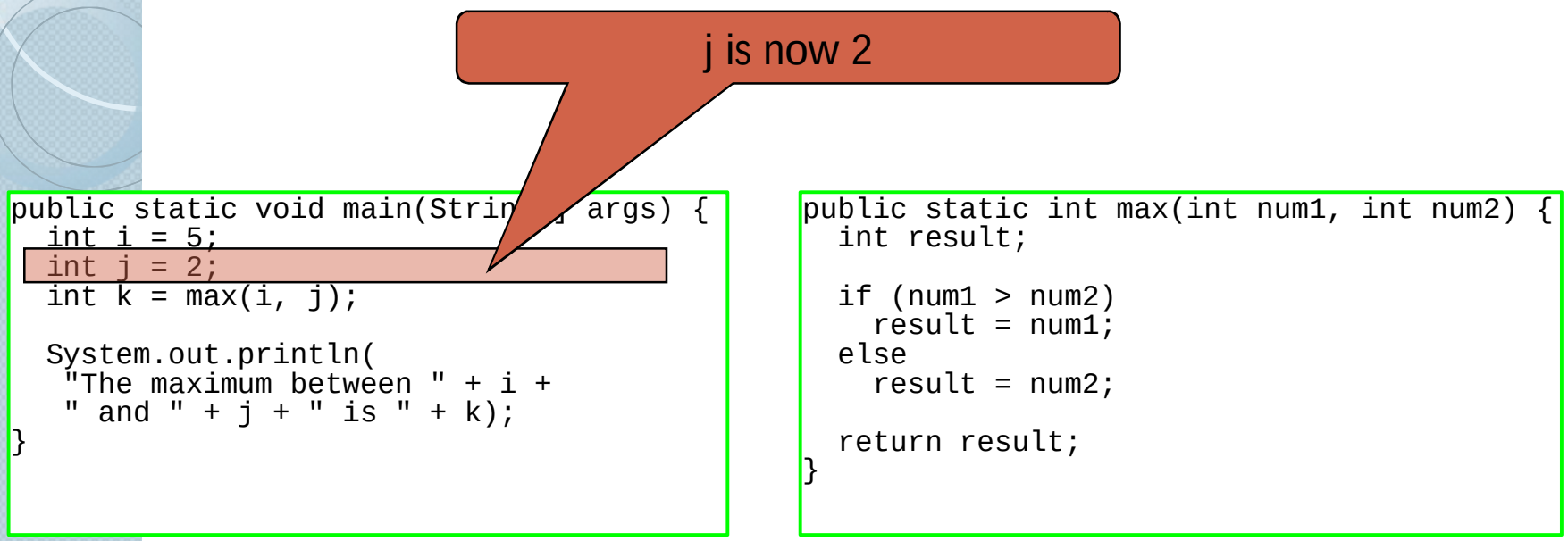
```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

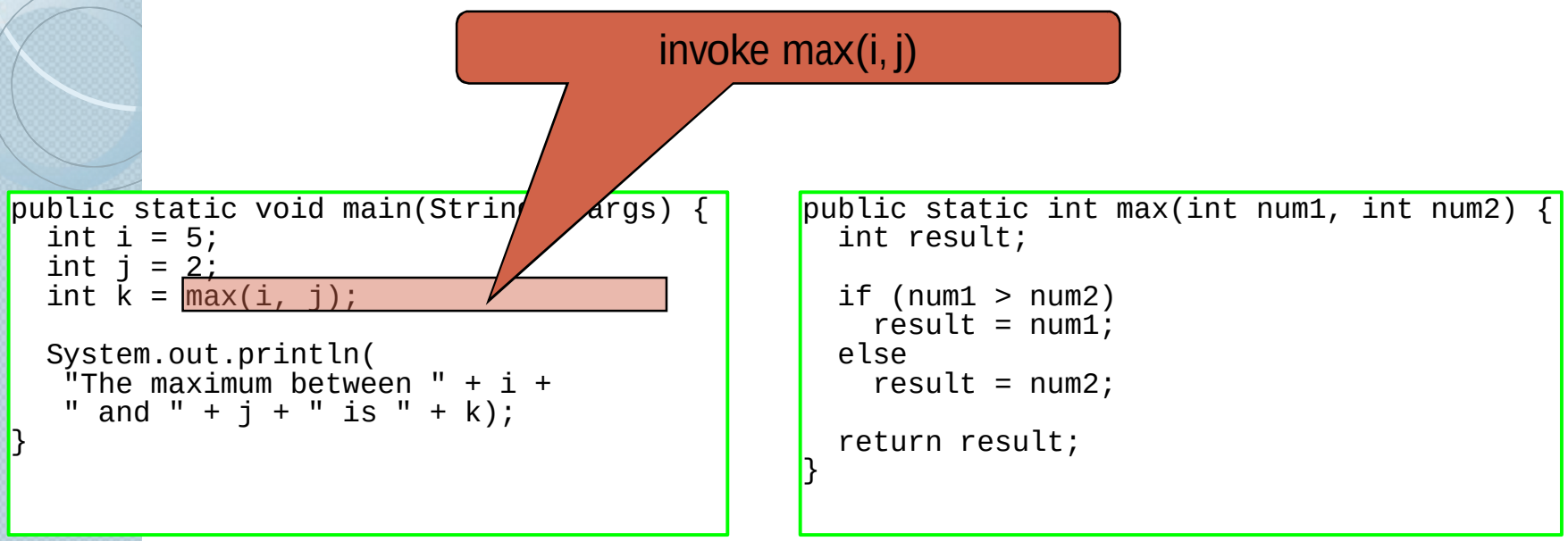
animation

Trace Method Invocation



animation

Trace Method Invocation



animation

Trace Method Invocation

invoke max(i, j)
Pass the value of i to num1
Pass the value of j to num2

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

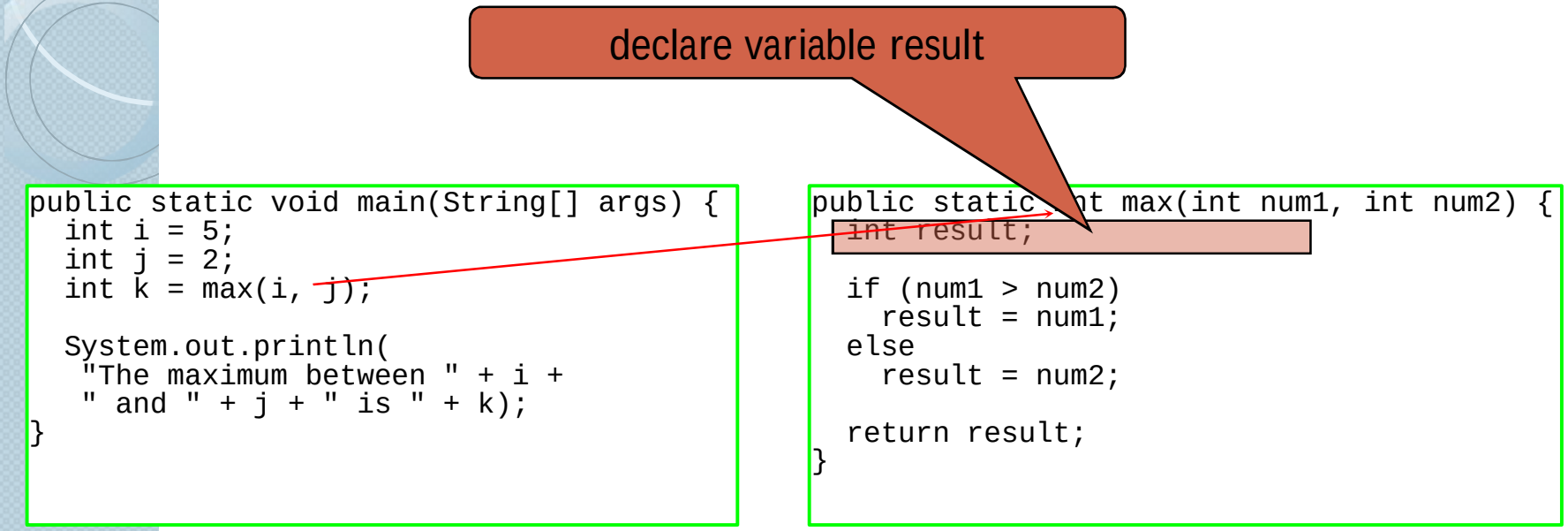
```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

animation

Trace Method Invocation



animation

Trace Method Invocation

(num1 > num2) is true since num1 is 5 and num2 is 2

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

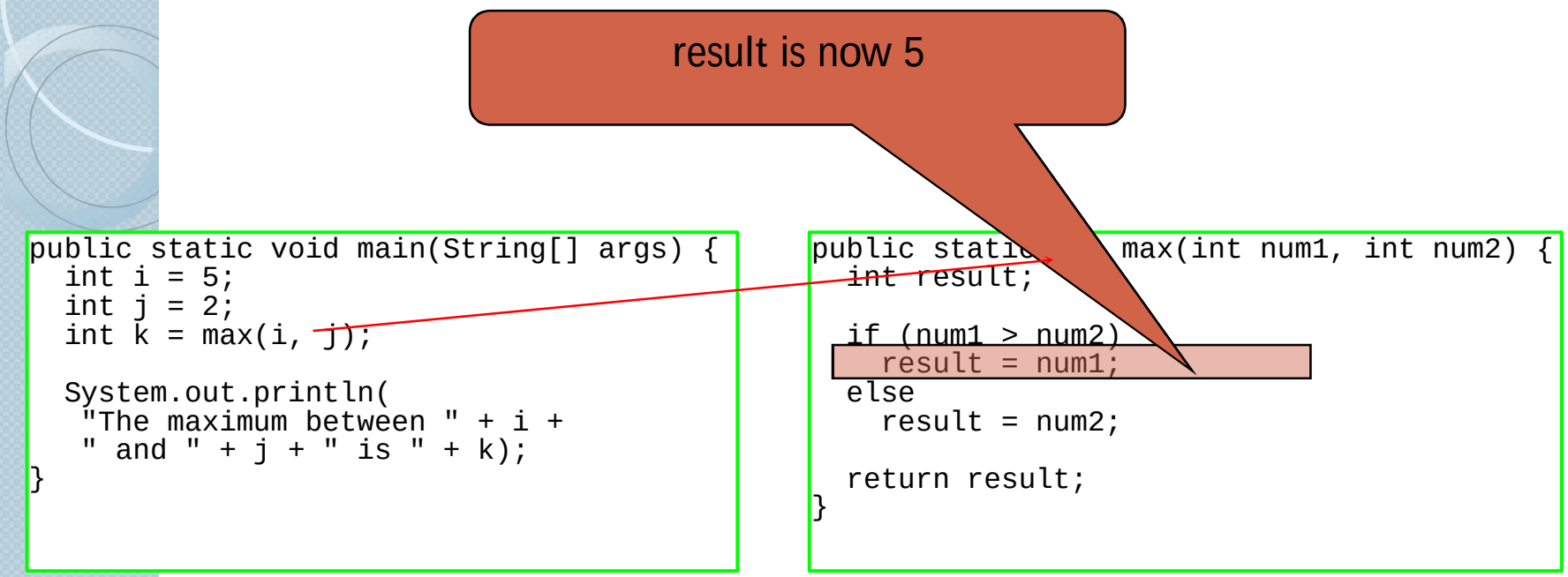
```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

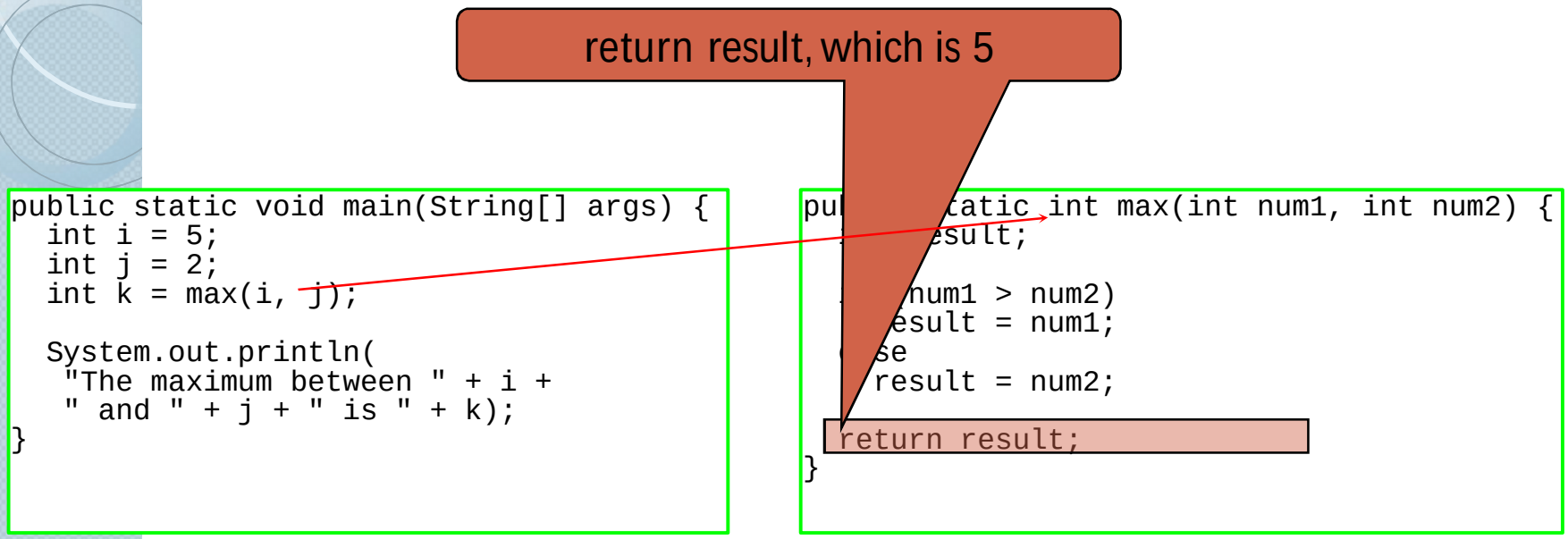
animation

Trace Method Invocation



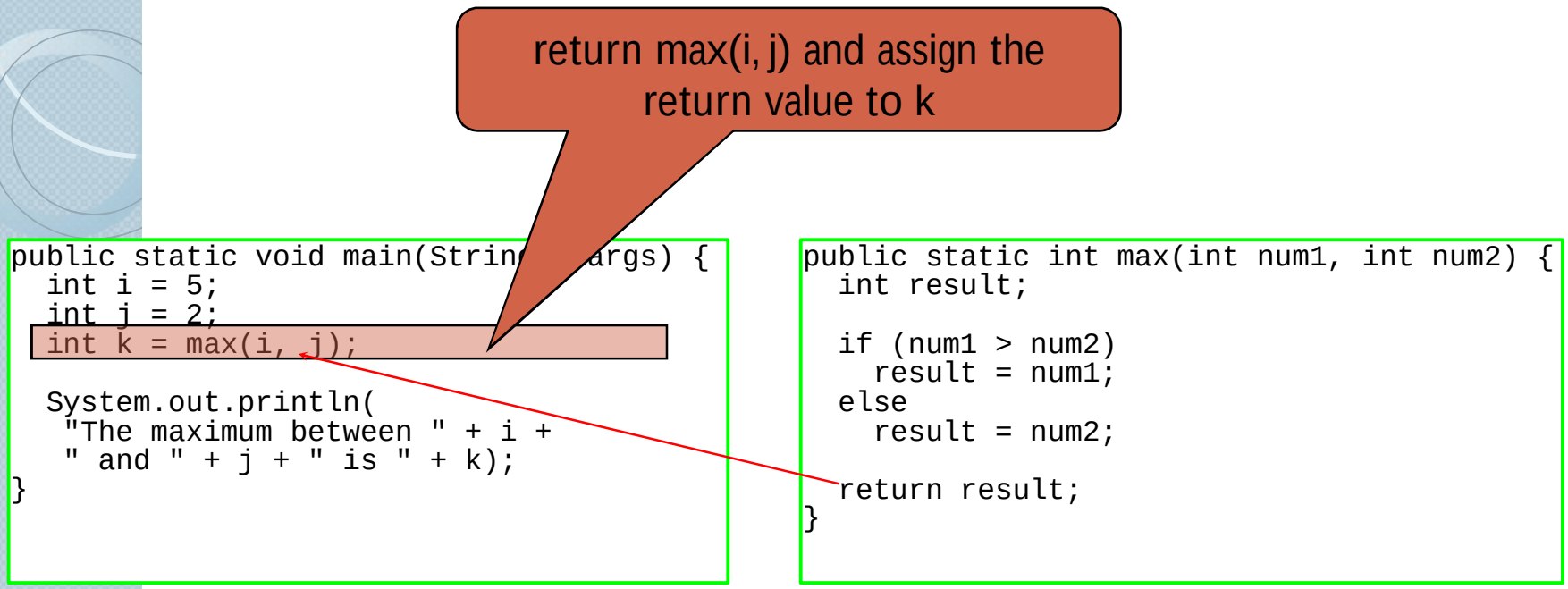
animation

Trace Method Invocation



animation

Trace Method Invocation



animation

Trace Method Invocation

Execute the print statement

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

Instance Methods

- An instance method:
  - represents one "behavior" for a particular kind of object.
  - is a method that is sent to an instance of a class.
  - is the most common kind of method since it represents interaction with particular objects
  - must not be declared static.
- Most methods that you'll write are instance methods since they operate on instances of the object for which you are defining the method.

Instance Methods

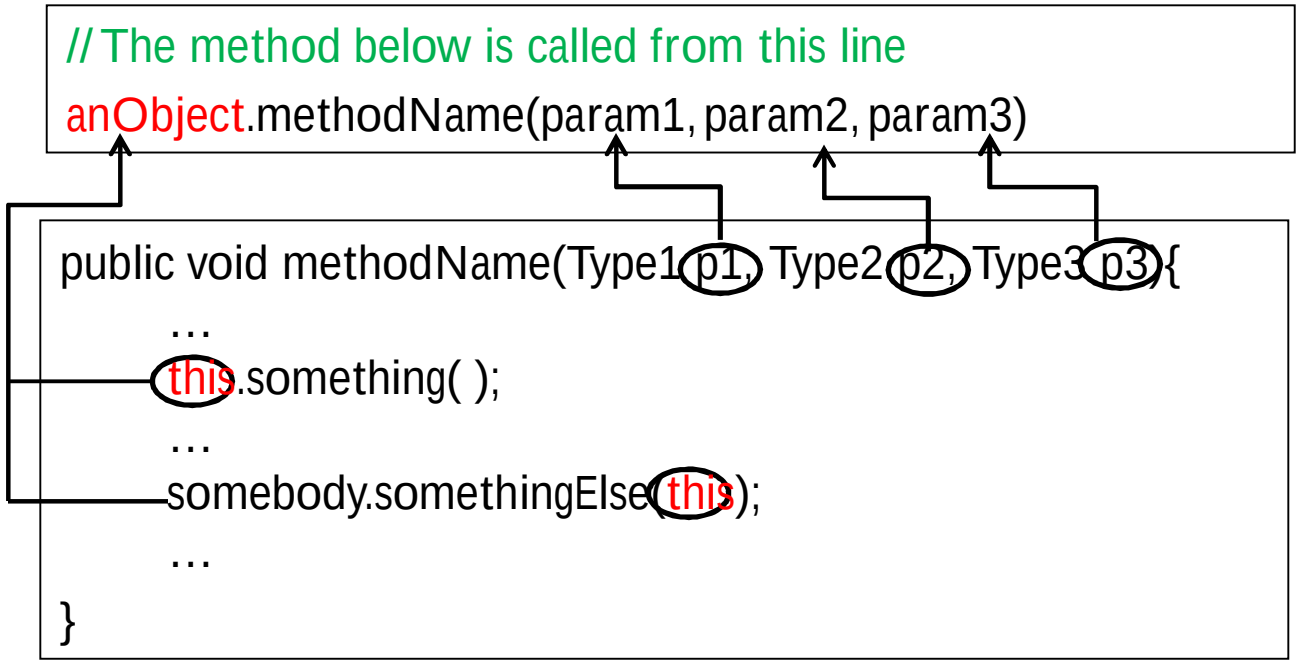
- So in summary, when using instance methods, they can be called two ways:

<methodName>() //when the method with name <methodName> is declared in this class

<instance>.<methodName>() //when the method with name <methodName> is declared in a different class

Instance Methods (Cont.)

- Essentially, the keyword this represents the receiver of the method. That is, when we are writing code inside a method, we can use the word this to represent the object that will be receiving the message which we are defining.
- In essence, the keyword this will represent the instance itself when the method is executed. Consider the example below in which the method called methodName is called from somewhere in your application:



Instance Methods (Cont.)

- Notice that the this keyword simply acts like a "special" parameter that refers to the object who has received the message.
- So, this actually depends on the context of where this code lies. It may represent completely different kinds of objects.
- For example, when we use this in a non-static method of the Circle class, this will represent an instance of a Circle. If we were to use this inside of an instance method in a Rectangle class, then this would represent an instance of Rectangle.
- In fact, if we leave out the this keyword, java assumes that it is the receiver's instance variable that we are trying to access, so we don't really need it.

Instance Methods (Cont.)

- The MOST IMPORTANT use of this is when we would like to:
  - send the receiver object to another method, or
  - return the receiver object

```
public class Client{
   public void purchase(Bond x){
            x.setClient(this);
   }
}
```

```
public class Bond{
   public void setClient(Client c){
            ……
   }
}
```

- For example, a Client may purchase a Bond and the Client information may need to be stored with the Bond:
- Notice that we are forced to use the keyword this because it is the only way we have to refer back to the receiver Client being stored in the Bond.

Instance Methods (Cont.)

- In the case where the return type of a method is an object, null may be returned. In fact, null is often returned as a result from "searching" methods to indicate that the search was unsuccessful:

```
public Object find(String x){
        // look for x
         ...
        // if it was not found, there is no answer
        return null;
}
```

- Example: EnlargeTester
- Example: FitsInsideTester

Class Methods

- We have just examined the creation of instance methods that are used to access/modify our own objects. Now lets look at how to create static class methods.
- A class method (i.e., static method):
  - is a method that is sent to a class object (although java allows us to send it to an instance as well … sort of weird though)
  - represents shared behavior that does not depend on particular instances
  - must be declared as static
- A class method is often used to represent a function which performs some computation and is independent of a particular instance.

Class Methods (Cont'd)

- JAVA already has some useful static functions which we can use in our programs (we have seen a few of these in the Math class).
- Example: Consider the following code which converts a centigrade temperature to Fahrenheit:

```
System.out.println ("Enter a centigrade temperature:");
int cent = new Scanner(System.in).nextInt();
int fahr = cent * 9 / 5 + 32;
System.out.println(" C is " +cent + "F is "+ fahr);
```

- This code forces the user to type in the centigrade temperatures each time.

Class Methods (Cont.)

- We can simply create our own class called Converter and make a message called centigradeToFahrenheit().

```
public class Converter{
    public static void centigradeToFahrenheit (){
            System.out.println ("Enter a centigrade temperature:");
            int cent = new Scanner(System.in).nextInt();
            int fahr = cent * 9 / 5 + 32;
            System.out.println(" C is " +cent + "F is "+ fahr);
    }
}
```

- Note that the method requires a single integer parameter (i.e. the centigrade value) and returns a single integer value (i.e., the converted temperature).

Class Methods (Cont.)

- Since we have defined the method as static, it is a "class" method and we can call it as follows:

  Converter.centigradeToFahrenheit();

- In fact, if we want to call it from any other method within this same class, we can just do this: centigradeToFahrenheit()

- since Java assumes by default that it is a class method for this class if we do not specify the class. Now, in order to see if it works, we need to write some test code:

```
public class ConverterTester{
    public static void main(String args[]){
            Converter.centigradeToFahrenheit());
            Converter.centigradeToFahrenheit());
            Converter.centigradeToFahrenheit());
    }
}
```

Class Methods (Cont.)

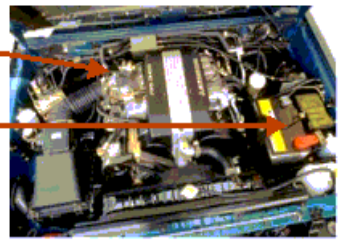
- So, class methods are much like instance methods in the way they work, except that class methods DO NOT require an instance in order to be called.
- So in summary, when using methods, they can be called three ways:

```
<methodName>()      //when the method with name <methodName> is declared
                    //in this class
<instance>.<methodName>()  //when the method with name <methodName> is
                           //declared in a different class
<classname>.<methodName>()  //when the method with name <methodName>
                            //is declared static in another class
```

3. Object Initialization and Method overloading

- When we create primitive variables, they always have some kind of initialized state (zero) before we assign them a value.
- Similarly, reference variables have an initial state of null before we assign values to them.
- However, when defining our own objects, we may want to specify more interesting default values, rather than zeros and nulls.
- In real life, upon creating an object we ALWAYS expect it to have some kind of initial state.

» fully assembled
» painted
» battery fully charged
» full tank of gas
» etc...



Object Initialization _Constructors

- In JAVA, using a constructor is the ONLY way to create an instance of your object.
- A constructor:
  - is a special kind of method that is ALWAYS called when an object is created.
  - is used to create an object and initialize its instance variables.
- If using a constructor is the ONLY way to create an instance of an object, how come we were able to create Circles, Rectangles, etc... without having to write a constructor method ?

Object Initialization _Constructors

- If no constructor is created for an object, then JAVA supplies a default constructor that will have behavior which will create the object, but leave the state un-initialized (i.e., zeros and nulls).
- By using new <classname>(), you are actually calling the default constructor for that class. Notice that the default constructor has no arguments (i.e., no parameters).
- Clearly, we can call the default constructor and then initialize the state directly.

Object Initialization _Constructors

- Constructors are methods that:
  - have same name as the class
  - have no return type
  - typically initialize values of all fields
  - are usually declared public
  - may have any number of parameters
- Note: We may not wish to supply parameters for all the fields:
  - some fields may not need to be specified by the user
  - allows user to specify what he/she knows and lets the program decide on other values

Object Initialization _Constructors

- Here is an example of a constructor for a BankAccount:

```
public BankAccount(String initName, int initAccountNumber, float initBalance)
{
        ownerName = initName;
        accountNumber = initAccountNumber;
        balance = initBalance;
}
```

- The parameter types for a constructor often match the field types (but their order is not important). One thing that IS important is that we SHOULD NOT use the same names for our parameters as our instance variables.

Object Initialization _Constructors

- For example, the following code would not work:

```
public BankAccount(String ownerName, int accountNumber, float balance)
{
        ownerName = ownerName;
        accountNumber = accountNumber;
        balance = balance;
}
```

- the constructor code WILL indeed compile ! However, it will leave the instance variables un-initialized.

Object Initialization _Constructors

- In fact, we could use the this keyword here to avoid ambiguity with variable names. Here is an example of how this can be used in the constructor method to avoid confusion with parameters having the same name:

```
public BankAccount(String ownerName, int accountNumber, float balance)
{
        this.ownerName = ownerName;
        this.accountNumber = accountNumber;
        this.balance = balance;
}
```

- Now the JAVA compiler knows that the left side is the instance variable and the right side is the parameter.

Object Initialization _Constructors

- For a BankAccount, we may assume that new bank accounts always have a balance of zero. Then we don't need to pass an initial balance as a parameter:

```
public BankAccount(String initName, int initAccountNumber)
{
        ownerName = initName;
        accountNumber = initAccountNumber;
        balance = 0.0f; //our own default value
}
```

- Notice that we have another constructor with different parameters.
- When there are more than one constructor, how does JAVA know which one we are calling ? This brings up the notion of overloading.

Method Overloading

- JAVA allows many methods to have the same name. We have already written two methods called area(), one for Circles, another for Rectangles. JAVA knew which one we were calling by the object that was using it. This is an example of what is called polymorphism (which we will discuss later).
- In fact, we are allowed to have two methods in the SAME class with the SAME name as long as they have a different set of parameters.

Method Overloading

- Overloading is often used to:
  - provide similar functionality with different types of data.
  - help keep code simpler (easy to remember method names).
- Consider some methods with the same name:

```
public double calculate (BankAccount account){
…
}
Public double calculate (Mortgage mtge, double interest){
…
}
Public double calculate (Lease aLease){
…
}
```

Method Overloading

- When called, how does JAVA know which one to use ?
  System.out.println(calculate(x));
- JAVA simply compares the name and list of parameter types to determine which one to call:
- if x is a BankAccount object, the first method is called.
- if x is a Lease object, the third method is called.

Method Overloading

- There cannot be two methods with the same name, parameter types and different parameter names:

```
public double calculate (BankAccount account){
…
}
Public double calculate (BankAccount x ){
…
}
```

- There also cannot be two methods with the same name and parameter list with different return types!

```
public double calculate (BankAccount account){
…
}
Public int calculate (BankAccount account ){
…
}
```

Method Overloading _ Constructor

- Multiple constructors: you may create many constructors but you should always have at least two. One of these should ALWAYS be a zero-argument/default constructor.
- Some valid constructors:

```
public BankAccount() {
    ownerName = "";
    accountNumber = 999999;
    balance = 0.0f;
}
public BankAccount(String initName) {
    accountNumber = LAST_ACCOUNT_NUMBER++;
    ownerName = initName;
    balance = 0.0f;
}
public BankAccount(String initName, float initBalance) {
    accountNumber = LAST_ACCOUNT_NUMBER++;
    ownerName = initName;
    balance = initBalance;
}
public BankAccount(String initName, int initAccountNumber, float initBalance) {
    ownerName = initName;
    accountNumber = initAccountNumber;
    balance = initBalance;
}
```

Method Overloading _ Constructor

- We can call each of these constructors as follows:

```
BankAccount myAccount, yourAccount, jimsAccount, marysAccount;
myAccount = new BankAccount();
yourAccount = new BankAccount("Shelly");
jimsAccount = new BankAccount("Jim", 100.0f);
marysAccount = new BankAccount("Mary", 478734, 100.0f);
```

- Notice that it is a good idea to give initial values to ALL of your object's instance variables. In the case where there is no initial value supplied as a parameter, you should choose a meaningful default value and "hard-code" it in the method.

Method Overloading _ Constructor

- When writing many constructors, often we have repeated code. We can have one constructor call another through a notion of chaining. We simply call one constructor from within another by using the this keyword as if it was a method name:

```
public BankAccount(String initName, int initAccountNumber, float initBalance) {
    ownerName = initName;
    accountNumber = initAccountNumber;
    balance = initBalance;
}
public BankAccount() {
    this("", 999999, 0.0f);
}
public BankAccount(String initName) {
    this(initName, LAST_ACCOUNT_NUMBER++, 0.0f);
}
public BankAccount(String initName, floatinitBalance) {
    this(initName, LAST_ACCOUNT_NUMBER++, bal);
}
```

Modifiers and Data Encapsulation

- When creating and defining an object, a main concern is to make sure that the code:
  - is as simple to use as possible,
  - is easily maintainable,
  - is as self-contained as possible, and
  - only reveals to other classes what is necessary.
- It is also good programming practice to NOT allow other classes (outside users of a class) to access or modify the instance variables of an object directly.

Modifiers and Data Encapsulation (Cont.)

- As we can see, OOP encapsulates both data AND behavior into objects. Each object should take advantage of the characteristic of data hiding.
- That is, the user of a class does not need to know how the object is actually represented, just how it is to be used. This is known as encapsulation .
- The idea is to simply hide the "details" of the object from general users of the object so that using the object becomes simpler.
- In JAVA, we can simplify and protect our objects by simplifying and protecting both its state and behavior. Both state and behavior for our object are considered "members" of its class.

Modifiers and Data Encapsulation (Cont.)

- A modifier represents the "visibility" or "permissions" for using a class or one of its members. Certain modifiers make sense for certain members.
- We use modifiers to:
  - simplify the outsider's view of our code
  - to prevent outsiders from accessing/modifying internal components that we deem to be private

Modifiers and Data Encapsulation (Cont.)

- There are many keyword modifiers in JAVA, each with its own meaning. We simply need to use these keywords when declaring variables, methods and classes. When we declare a variable, here is the format we follow:

```
[public, private, protected] [static] [final] DataType variableName;
```

- Everything in the square brackets is optional. Note also that only one of public, private or protected is allowed at a time. These modifiers are described below.
- When declaring a method, here is the format we follow:

```
[public, private, protected] [static] [abstract, final] ReturnType methodName
(Type1 name1, Type2 name2, ...) {...}
```

- When declaring a class, here is the format we follow:

```
[public] [abstract] [final] class className {...}
```

Modifiers and Data Encapsulation (Cont.)

- Here is a list of some valid access modifiers along with a brief description of how they are used for classes and members:

* At this point, we will leave out the explanations of the abstract, protected and final keywords and get back to them when we discuss inheritance.

Modifier	Class	Member (instance variable or method)
None	Accessible only within its package	Accessible only within its package
Public	Accessible anywhere that its package is	Accessible anywhere that its class is
private	n/a	Accessible only within its own class

Modifiers and Data Encapsulation (Cont.)

_ Class Modifiers

- How are the modifiers used for classes? Note that when we declare a class, we usually do this:

```
public class <classname> {
}
```

- By using the word public, we are allowing anyone to use the class (as long as they can see the package (or directory) that contains the class.
- We could have also left out the word public as follows:

```
class <classname> {
}
```

- By leaving it out, we are allowing default access ... which means that they are accessible from any classes which are defined in the same physical directory that this class is situated.

Modifiers and Data Encapsulation (Cont.)

_ Class Modifiers

- The import command in JAVA is used to tell the compiler that you'd like to use classes/methods or variables that are defined within some package.
- We have seen it used to import java.util.Scanner which indicates that we were using public classes/methods within the standard java.util.Scanner class.
- If however, the classes/methods required lie within the same directory as the class using it, then import is not necessary.

Modifiers and Data Encapsulation (Cont.) _ Class Modifiers

- Although we cannot normally make a class private, JAVA does allow us to make inner classes.
- An inner class is a class that is contained completely within another class. This kind of class is saved along with the containing class but it is not accessible outside the class.
- JAVA will actually compile this class separately and you will see a .class file generated which contains a $ in the name.

Modifiers and Data Encapsulation (Cont.) _ Class Modifiers

```
public class <OuterClassname> {
    ...
    private class <InnerClassname> {
        ...
    }
    ...
}
```

- JAVA allows only one public class per file. Nobody may access the inner class from anywhere other than the class that contains it.

Modifiers and Data Encapsulation (Cont.) _ Instance Variable Modifiers

- Now what about variables ? Well, the public modifier works the same way as with classes.
- While freedom to access/modify anything from anywhere seems like a friendly thing to do, it is certainly dangerous. So "rules of thumb":
  - rarely use public on instance variables
  - use public on methods when you feel that external users of your class would need/want to use the method
- Consider the private modifier. If we declare a method or instance variable as private, this restricts its access to be that of only the class that defines it.
- Therefore, we are actually hiding (or encapsulating) the details of our object from others. This is known as encapsulation.

Modifiers and Data Encapsulation (Cont.) _ Class Modifiers

- Keep in mind the purpose of this idea: to protect our inner parts.
- As an example, we can directly access the balance of a BankAccount. This is clearly undesirable since there is little protection.

```
BankAccount myAccount = new BankAccount(432883,“Lucy");
System.out.println("The balance is $" + myAccount.balance);
myAccount.balance = 1000000.0f;

System.out.print("I just modified the account balance. ");
System.out.println("The balance is now $" + myAccount.balance);
```

Modifiers and Data Encapsulation (Cont.) _ Class Modifiers

- Now to avoid this, we can simply declare the balance (as well as the other fields) to be private.

```
public class BankAccount {
    private String   ownerName;
    private int      accountNumber;
    private float    balance;
    ...
}
```

- Now we cannot access these variables outside the class. A compile error will result.

```
myAccount.balance = 1000000.0f;    // will no longer compile
```

Modifiers and Data Encapsulation (Cont.) _ Class Modifiers

- Get / Set Methods:

  If I hide an object's state from everyone

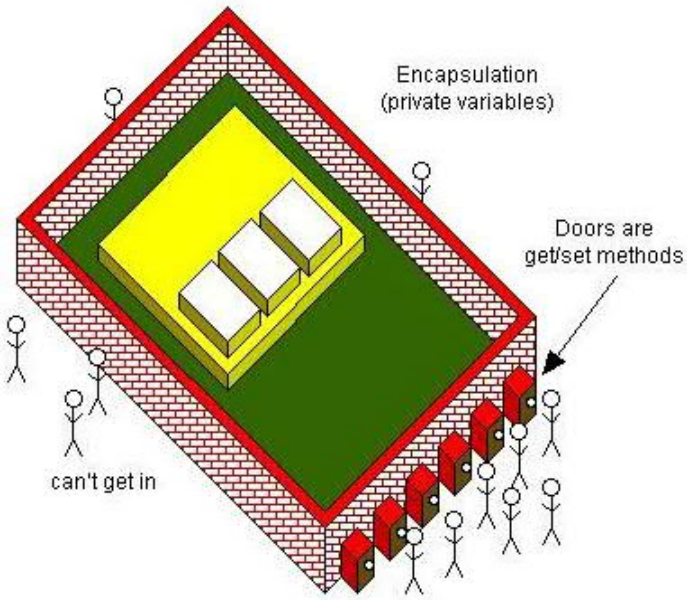
  – how are they going to know what the object is made up of ?

  – How can anyone possibly use a class without knowing what it is made of ?

  Well, just like a walled city, we'll have to mak doors/gates to allow access. We do this by making two types of methods:

  – get method (getter) to access each field

  – set method (setter) to modify each field



Modifiers and Data Encapsulation (Cont.) _ Class Modifiers

- The set methods let us put information into our object, the get methods let us look at what is in there:
- Get methods have:
  - a name that corresponds to the field's name: get<FieldName>
  - a return type that matches the field's type.
  - code that merely returns the field's value
- Normally, ALL get methods have this format:

```
public <fieldType> get<FieldName>() {
    return <fieldName>;
}
```

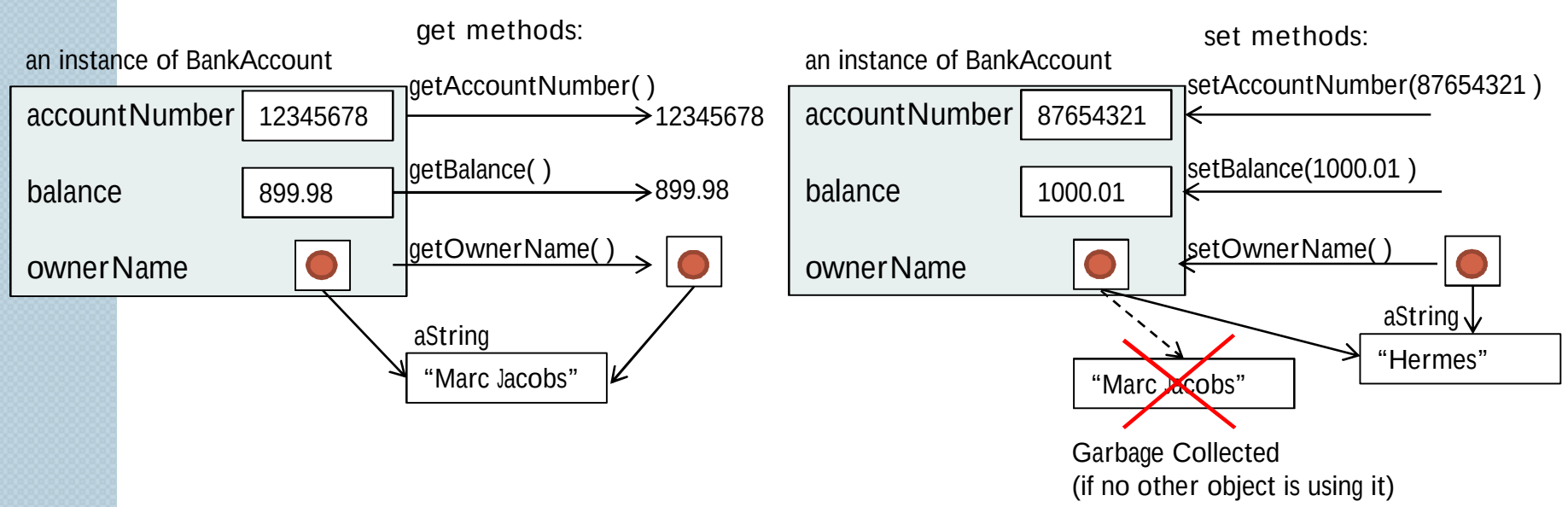
Modifiers and Data Encapsulation (Cont.) _ Class Modifiers

- Similarly, set methods have:
  - a name that corresponds to the field's name: set<FieldName>
  - no return type (declared as void).
  - a single parameter which is same type as the field's type.
  - code that merely assigns the parameter value to the field.
- Normally, ALL set methods have this format:

```
public void set<FieldName>(<fieldType> <paramName>) {
   <fieldName> = <paramName>;
}
```

Modifiers and Data Encapsulation (Cont.) _ Class Modifiers

- Just to make sure that you understand the purpose of get/set methods, remember this:
  - get methods let you access an object's fields (like doorways to the encapsulated city)
  - set methods let you modify an object's fields (also through doorways)



Modifiers and Data Encapsulation (Cont.) _ Class Modifiers

- Example: for our BankAccount with fields ownerName, accountNumber and balance, here is what the get and set methods look like

```
public String getOwnerName() {
    return ownerName;
}
public int getAccountNumber() {
    return accountNumber;
}

public float getBalance() {
    return balance;
}
```

```
public void setOwnerName(String
newOwnerName) {
    ownerName = newOwnerName;
}
public void setAccountNumber(int
newAcctNum) {
    accountNumber = newAcctNum;
}

public void setBalance(float newBalance)
{
    balance = newBalance;
}
```

Modifiers and Data Encapsulation (Cont.) _ Class Modifiers

- How we can use get/set methods with the dot operator:

```
BankAccount myAccount = new BankAccount();
myAccount.setOwnerName("Joe Cool");
myAccount.setAccountNumber(274834);
myAccount.setBalance(1500.00f);

System.out.print("Owner name is: ");
System.out.println(myAccount.getOwnerName());
System.out.print("Account Number is: ");
System.out.println(myAccount.getAccountNumber());
System.out.print("Balance is: $");
System.out.println(myAccount.getBalance());
```

Modifiers and Data Encapsulation (Cont.) _ Class Modifiers

- Encapsulation has advantages:
  - makes life simpler for user of class:
    - don't have to understand "guts" of the object being used.
    - allows us to treat object as a "black box".
  - changes to object's state do not affect outside world.
  - prevents uninformed users from making direct state changes that can invalidate an object.

Modifiers and Data Encapsulation (Cont.) _ method Modifiers

- When the public modifier is used with methods, others can freely use this method from anywhere.
- If we declare a method as private, we cannot use this method from any class other than the class in which it is defined.
- Usually, private methods are called helper methods since they are often created for the purpose of reducing the size of a larger public method or when code needs to be used by several methods.

Modifiers and Data Encapsulation (Cont.) _ Class Modifiers

- Here is an example of breaking up the problem of solving a jigsaw puzzle into helper methods :

```
public class Person {
    public void solvePuzzle() {
        solveBorders();
        solveMainPicture();
        solveBackground();
    }
    private void solveBorder() {      ...      }
    private void solveMainPicture() {      ...     }
    private void solveBackground() {      ...     }
}
```

Note that the helper methods are private since they should not be used outside the scope of the class. These methods only exist so that we don't have to write a large solvePuzzle() method.

Bank Account Case Study (Cont.)

- It is now time to make a complete class. We will make a complete BankAccount object by gathering together all that we have learned so far such as:
- Step 1: Make a class called Bank. It should maintain it's owner name, account number and account balance.
- Step 2: Write appropriate get/set/constructor methods.
- Step 3: some interesting methods ? Well we should decide what kinds of operations can be performed on this kind of object.
    - That is, with a BankAccount, we can deposit money, withdraw money etc....
- Step 4: toString() method

Bank Account Case Study (Cont.)

- Step 5: Example Methods
  - To aid with testing, we can create special `example` methods which are used to create "examples" of `BankAccounts`. Example methods:
    - provide the user of a class with a quick way of obtaining a typical example of an instance of that class
    - are used in testing so as to simplify the code by "hiding" the code that creates and initializes an object to some default values.
    - are `static` methods

Bank Account Case Study (Cont.)

Step 6: Adding a Class Variable

- Looking at the BankAccount code, we notice something unrealistic. When creating a new bank account, we probably should not be allowed to specify the account number.
- Let us assume that the first customer gets 000001, the second gets 000002, the third 000003 and so on.
- We'll keep a counter called LAST_ACCOUNT_NUMBER which will store the account number that was given out last. The next customer will get one more than this.
- How do we do this ? Well, just make a class variable (i.e., static) called LAST_ACCOUNT_NUMBER which has an initial value of 0. Then, when a new BankAccount is created, give it an account number which is one more than the LAST_ACCOUNT_NUMBER and increment it for the next time.

Literals and Enumerated Types

- In version 1.5 of JAVA, a new type has been added called an enumerated type. Essentially, an enumerated type is a fixed set of "symbols" that are grouped together. The format is as follows:
  - <modifier> enum <TypeName>{ value1, value2, value3, ... };
- Here, the type name should be capitalized and should be different from any class names being used. The values are essentially fixed textual symbols commonly written in uppercase characters. Here are some examples:
  - public enum Sex {MALE, FEMALE};
  - public enum Size {XS, SMALL, MEDIUM, LARGE, XL};

Literals and Enumerated Types (Cont.)

- public enum Weekday {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY};
- public enum CompassDirection {NORTH, NORTH_EAST, EAST, SOUTH_EAST, SOUTH, SOUTH_WEST, WEST, NORTH_WEST};

- These types can then be stored in variables and used as constant literal values by specifying the enum's TypeName followed by the dot operator and then the particular desired value;
  - Sex defaultSex = Sex.MALE;
  - Size orderSize = Size.MEDIUM;
  - CompassDirection homePosition = CompassDirection.NORTH_WEST;
- Example: EnumTest.java

Case Study

- A popular game of chance is dice game known craps, which is played in casinos and back alleys throughout the world. We will develop a revised “craps” game with the following rules: You roll two dice. Each dice has maximum 10 faces (You may initialize the face numbers). After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, you win. If the sum is 2, 3, or 12 on the first throw (called “craps”), you lose (i.e., the “house” wins). If the sum is any other number on the first throw, that sum becomes your “point”. To win, you must continue rolling the dice until you “make your point” (i.e., roll that same point value). You lose by rolling a 7 before making your point.

Case Study

- Define a Dice class. Each dice should maintain a final variable NUM_SIDES keeping the number of faces, and a faceValue showing the upward face value. Define the appropriate get/set/constructor and toString() methods. Write a method called roll() that randomly generates the upward face value within a range from one to NUM_SIDES.

Case Study

- Define a Craps class. The attributes of this class are two dice objects, the game status of the player (i.e. won, lost, or continue, using enum to define the game status), and the sum of two upward face values. The methods are:
  - ✓ a constructor that allows users of the class to initialize the two dice objects and specify the number of sides on each die separately. In other words, the constructor has two input parameters. For instance, the first dice could be a six-sided dice and the second one a ten-sided dice.
  - ✓ a toString method that indicates the game status.
  - ✓ a method called rollBoth() that rolls both dice, and calculates the sum of the upward face values on the two dice.
  - ✓ A method called testYourLuck() that determines you win/lose the game according to game rule.
- Define a CrapsTest class to test your program.

Equality and Identity (Cont.)

- Two objects are equal if they are of the same type and have the same value. That is, if the values of the instance variables of the two objects are the same, then the objects are equal.
- Two objects are identical if and only if they are the exact same object.
- Consider this code:

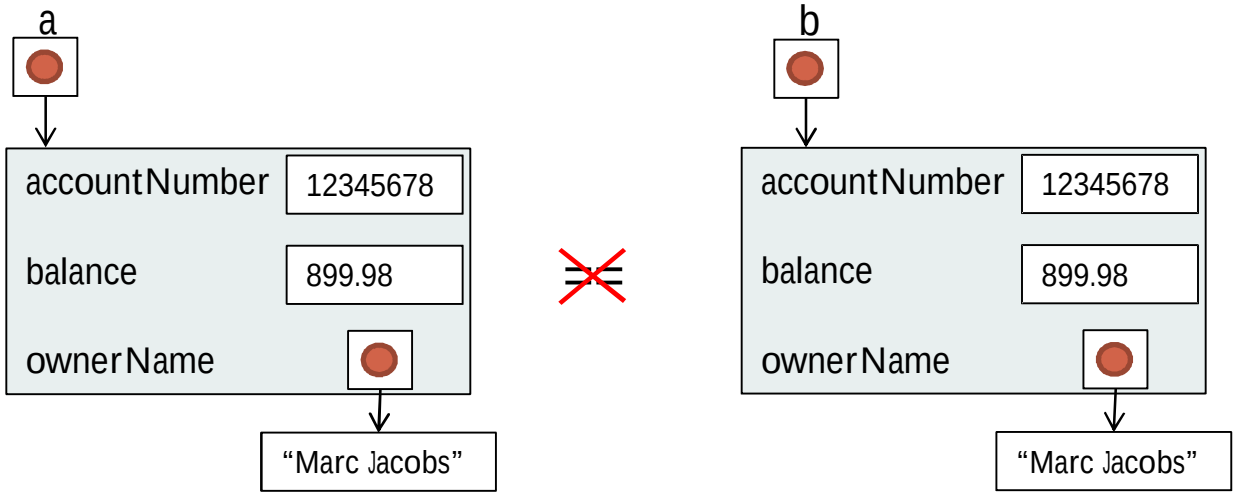
```
BankAccount a = new BankAccount("Marc Jacobs", 899.98f, 12345678);
BankAccount b = new BankAccount("Marc Jacobs", 899.98f, 12345678);
```

- If we want to ask whether or not these two accounts are equal, intuitively we'd write: if (a == b) { ... }

```
BankAccount a = new BankAccount("Marc Jacobs", 899.98f, 12345678);
BankAccount b = new BankAccount("Marc Jacobs", 899.98f, 12345678);
if (a == b) {
   System.out.println("YES");
}
else {
   System.out.println("NO");
}
```

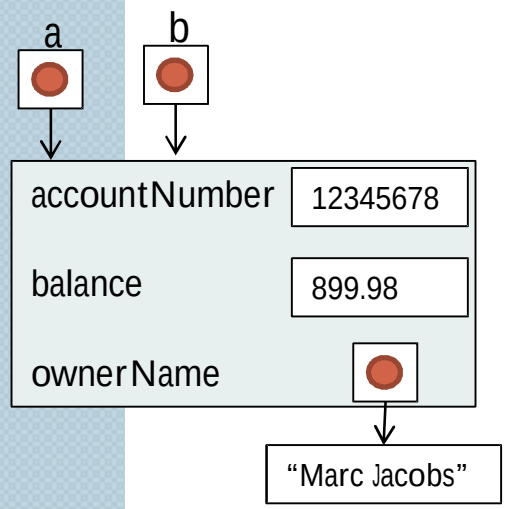
Equality and Identity (Cont.)

- However, in JAVA, when the == operator is used with objects, it means identity … not equality. It would only return true if the two variables referred to (or pointed to) the same exact object.



Equality and Identity (Cont.)

- We can clearly see that a and b are pointing to different objects. It seems that whenever we make a new instance of an object, we actually get a unique object every time, thus that new object can only be identical to itself. What kind of code would allow (a == b) to return true ? Well, we'd have to have code that allows the two variables to refer to the same object:




```
BankAccount a = new BankAccount(“Marc Jacobs", 899.98f, 12345678);
BankAccount b = a;
if (a == b) {
    System.out.println("YES");
}
else {
    System.out.println("NO");
}
```

Equality and Identity (Cont.)

- So how do we check for equality ? Well, for objects, JAVA allows us to use the method called equals().
All objects understand the equals() method. Here is what we can do:

```
BankAccount a = new BankAccount("Bob", 100.00f, 123456);
BankAccount b = new BankAccount("Bob", 100.00f, 123456);
if (a.equals(b)) {
   System.out.println("YES");
}
else {
   System.out.println("NO");
}
```

- If you try this code (which returns NO again!) Well by default, the equals() method does essentially the following:

```
public boolean equals(Object x) {
   return this == x;
}
```

Equality and Identity (Cont.)

- So, we see that the default equals() method simply returns a boolean indicating whether or not the two objects are identical.
- As it turns out, there is a better equals() method for standard JAVA classes. For example, Strings, Dates, Vectors etc... all have their own versions of an equals() method that does a proper comparison.
- If we want to test whether two objects of our own classes are identical, we must also write our own equals() method.

```
public boolean equals(Object x) {
   // First make sure the two objects are of the same class
   if (!(x instanceof BankAccount)) {
      return false;
   }
   // Now type-cast the incoming object to a BankAccount object
   BankAccount b = (BankAccount) x;
   // Now check for equality
   return (this.getAccountNumber() == (b.getAccountNumber());
}
```

Equality and Identity (Cont.)

- Why you should always write an equals() method for your own classes?
  - Even if you are not planning to compare your objects directly in your code, there are some methods in JAVA that make use of the equals() method.
  - For example, if you want to add your objects to any kind of collection (e.g., Vectors, ArrayLists, Hashtables, HashMaps, HashSets etc...) then you MUST implement the equals() method so that the methods in these collection classes work properly.

Equality and Identity (Cont.)

- The remove(), contains(), containsKey() methods are just three of the standard JAVA methods which make use of the equals() method.
- If you don't write an equals() method, then your code may not work properly if you make use of these Collection class methods. JAVA will certainly find the object since it will use the default equals() method which checks for identity.

- Identity Example: AppleTester.java

References

- Carleton University COMP 1005 and 1006:
  http://www.scs.carleton.ca/~lanthier/teaching/COMP1405/Notes/
  http://www.scs.carleton.ca/~lanthier/teaching/COMP1406/Notes/
- Armstrong Atlantic State University
  http://www.cs.armstrong.edu/liang/intro10e/
- Java Sun tutorial:
  http://docs.oracle.com/javase/tutorial/
- University of Chicago:
  http://people.cs.uchicago.edu/~asiegel/courses/cspp51036/
- MIT opencourseware
  http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/