eC Academy

Realize Your Dreams

AP Computer Science A Review

Week 3: Classes and Objects

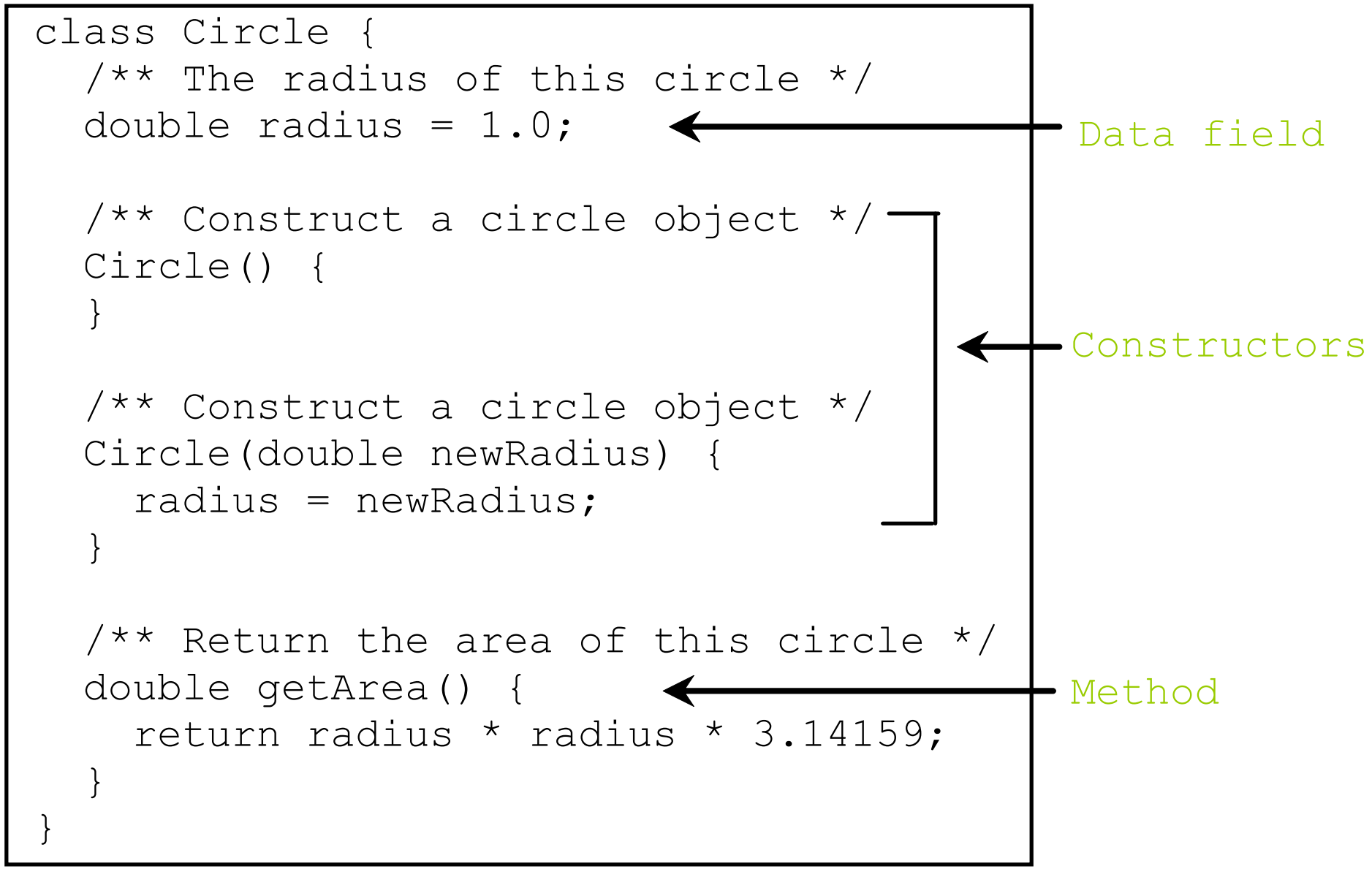
DR. ERIC CHOU
IEEE SENIOR MEMBER

Topics

- Objects and classes
- Data encapsulation
- References
- Keywords public, private, and static
- Methods
- Scope of variables

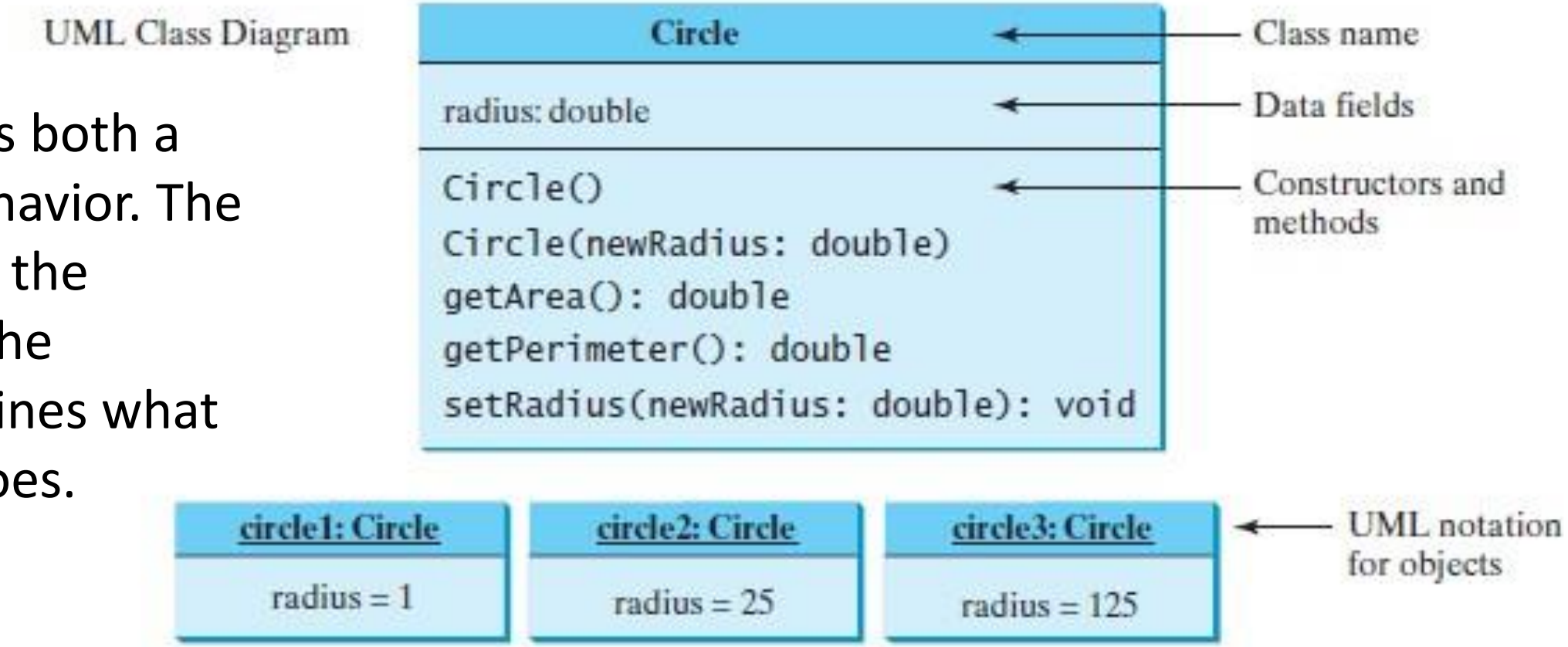
SECTION 1

Basic Class and Objects


class Circle {
/** The radius of this circle */
double radius = 1.0;
/** Construct a circle object */
Circle() {
}
/** Construct a circle object */
Circle(double newRadius) {
radius = newRadius;
}
/** Return the area of this circle */
double getArea() {
return radius * radius * 3.14159;
}
}
Data field
Constructors
Method

Objects: UML Class/Object Diagram

An object has both a state and behavior. The state defines the object, and the behavior defines what the object does.



Constructors, cont.

A **constructor** with no parameters is referred to as a *no-arg constructor*. If no constructor is given, default one will be used.

- Constructors must have the same name as the class itself.
- Constructors do not have a return type—not even void.
- Constructors are invoked using the new operator when an object is created. Constructors play the role of initializing objects.

Creating Objects Using Constructors

```
new ClassName();
```

Example:

```
new Circle();

new Circle(5.0);
```

Declaring Object Reference Variables

To reference an object, assign the object to a reference variable. For an object, you may have as many reference variable (pointer) as you wish.

To declare a reference variable, use the syntax:

```
ClassName objectRefVar;
```

Example:

```
Circle myCircle;
```

Declaring/Creating Objects in a Single Step

```
ClassName objectRefVar = new ClassName();
```

Example:

```
Circle myCircle = new Circle();
```

Accessing Object's Members
(Properties or Methods)

Referencing the object's data:

```
objectRefVar.data
```

e.g., `myCircle.radius`

Invoking the object's method:

```
objectRefVar.methodName(arguments)
```

e.g., `myCircle.getArea()`

Reference Data Fields

The data fields can be of reference types. For example, the following Student class contains a data field name of the String type.

```
public class Student {
  String name; // name has default value null
  int age; // age has default value 0
  boolean isScienceMajor; // isScienceMajor has default value false
  char gender; // c has default value '\u0000'
}
```

The null Value

- If a data field of a reference type does not reference any object, the data field holds a special literal value, **null**.
- The default value of a data field is **null** for a reference type, 0 for a numeric type, false for a boolean type, and '\u0000' for a char type. However, Java assigns no default value to a local variable inside a method.

Default Value for a Data Field

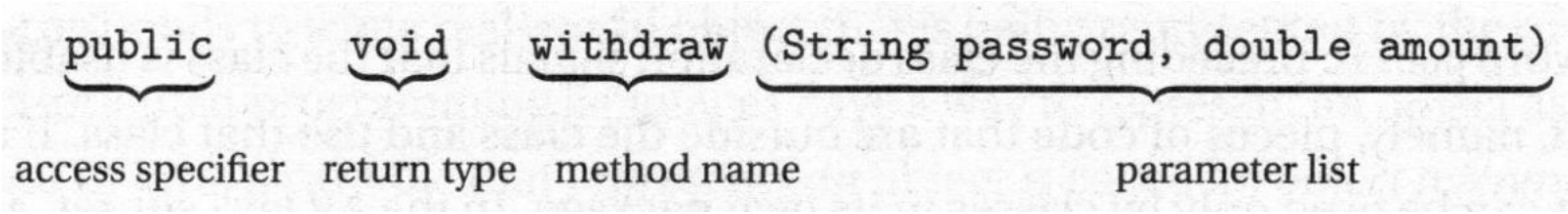
```
public class Test {
  public static void main(String[] args) {
    Student student = new Student();
    System.out.println("name?"+student.name);
    System.out.println("age?"+student.age);
    System.out.println("isScienceMajor?"+student.isScienceMajor);
    System.out.println("gender?" + student.gender);
  }
}
```

SECTION 2

Methods

Headers

All method headers, with the exception of constructors (see on the next page) and static methods (p. 105), look like this:



NOTE

1. The access specifier tells which other methods can call this method (see the "Public, Private, and Static" section on the previous page).
2. A return type of void signals that the method does not return a value.
3. Items in the parameter list are separated by commas.

The implementation of the method directly follows the header, enclosed in a {} block.

Types of Methods

CONSTRUCTORS

- A constructor creates an object of the class. You can recognize a constructor by its name always the same as the class. Also, a constructor has no return type.
- Having several constructors provides different ways of initializing class objects. For example, there are two constructors in the **`BankAccount`** class.
  1. The default constructor has no arguments. It provides reasonable initial values for an object. Here is its implementation:

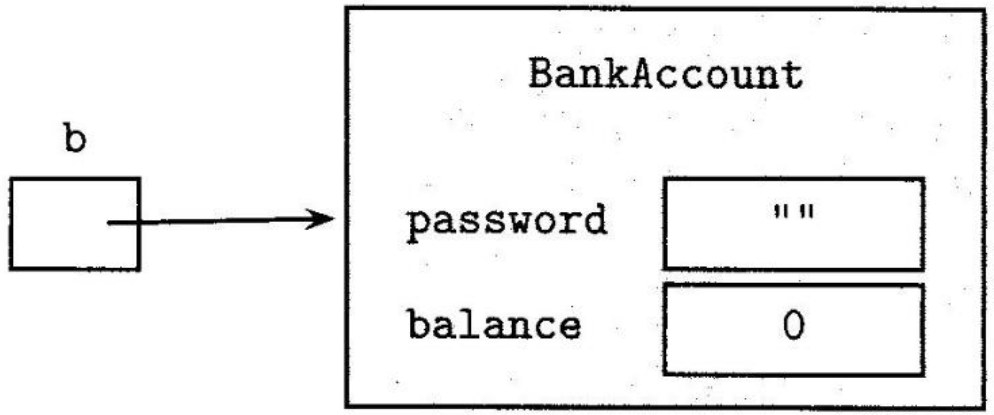
```
/** Default constructor.  Constructs a bank
account with default values. */
public BankAccount() {
  password = "";
  balance = 0.0;
}
```

Types of Methods

- In a client method, the declaration

```
BankAccount b = new BankAccount();
```

- constructs a `BankAccount` object with a balance of zero and a password equal to the empty string. The new operator returns the address of this newly constructed object.
- The variable *b* is assigned the value of this address-we say *"b is a reference to the object."* Picture the setup like this:



Types of Methods

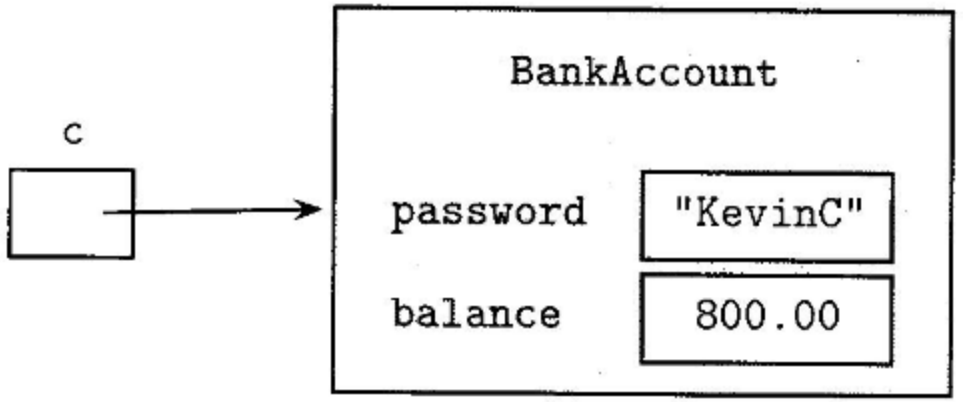
- 2. The constructor with parameters sets the instance variables of a BankAccount object to the values of those parameters.
- Here is the implementation:

```
/** Constructor. Constructs a bank account with
   * specified password and balance. */
public BankAccount(String acctPassword, double
acctBalance) {
   password = acctPassword;
   balance = acctBalance;
}
```

- In a client program a declaration that uses this constructor needs matching parameters:

Types of Methods

```
BankAccount c = new BankAccount("KevinC", 800.00);
```



NOTE

- band c are object variables that store the addresses of their respective `BankAccount` objects. They do not store the objects themselves (see "References" on p. 109).

Accessor

- An accessor method is a public method that accesses a class object without altering the object.
- An accessor returns some information about the object, and it allows other objects to get the value of a private instance variable.
- The BankAccount class has a single accessor method, getBalance(). Here is its implementation:

```
/** Returns the balance of this account. */
public double getBalance()
{ return balance; }
```

- A client program may use this method as follows:

```
BankAccount b1 = new BankAccount("MattW", 500.00);
BankAccount b2 = new BankAccount("DannyB", 650.50);
if (b1.getBalance() > b2.getBalance()) …
```

Mutator

- A mutator method changes the state of an object by modifying at least one of its instance variables. It is often a void method (i.e., has no return type). A mutator can be a private helper method within its class, or a public method that allows other objects to change a private instance variable.
- Here are the implementations of the deposit and withdraw methods, each of which alters the value of balance in the BankAccount class:

```
/** Deposits amount in a bank account with the given password. */
public void deposit(String acctPassword, double amount) {
if (!acctPassword.equals(password))
    /* throw an exception */
else
    balance += amount;
}

/** Withdraws amount from bank account with given password.
* Assesses penalty if balance is less than amount.
*/
public void withdraw(String acctPassword, double amount) {
    if (!acctPassword.equals(password))
        /* throw an exception */
    else {
        balance -= amount;
        //allows negative balance
        if (balance < 0)
        balance -= OVERDRAWN_PENALTY;
    }
}
```

Mutator Method

- A mutator method in a client program is invoked in the same way as an accessor: using an object variable with the dot operator. For example, assuming valid `BankAccount` declarations for b1 and b2:

```
b1.withdraw("MattW", 200.00);
b2.deposit("DannyB", 35.68);
```

Static Method

STATIC METHODS VS. INSTANCE METHODS

The methods discussed in the preceding sections constructors, accessors, and mutators-all operate on individual objects of a class. They are called instance methods.

A method that performs an operation for the entire class, not its individual objects, is called a static method (sometimes called a class method). **(shared property)**

Static Method

- The implementation of a static method uses the keyword static in its header. There is no implied object in the code (as there is in an instance method). Thus, if the code tries to call an instance method or invoke a private instance variable for this nonexistent object, a syntax error will occur.
- A static method can, however, use a static variable in its code. For example, in the Employee example on p. 102, you could add a static method that returns the employeeCount:

```
public static int getEmployeeCount()
{ return employeeCount; }
```

Static Method

- Here's an example of a static method that might be used in the `BankAccount` class. Suppose the class has a static variable `intRate`, declared as follows:

```
private static double intRate;
// The static method getInterestRate may be as follows:
public static double getInterestRate() {
    System.out.println("Enter interest rate for bank
    account");
    System.out.println("Enter in decimal form: ");
    intRate = ... ;
    return intRate;
}
```

Static Method

- Since the rate that's read in by this method applies to all bank accounts in the class, not to any particular BankAccount object, it's appropriate that the method should be static.
- Recall that an instance method is invoked in a client program by using an object variable followed by the dot operator followed by the method name:

```
BankAccount b = new BankAccount();
// invokes the deposit method for
b.deposit(acctPassword, amount);
// BankAccount object b
```

- A static method, by contrast, is invoked by using the class name with the dot operator:

```
double interestRate = BankAccount.getInterestRate();
```

Static Methods in a Driver Class

- Often a class that contains the main() method is used as a driver program to test other classes. Usually such a class creates no objects of the class.
- So all the methods in the class must be static. Note that at the start of program execution, no objects exist yet. So the main () method must always be static.
- For example, here is a program that tests a class for reading integers entered at the key board:

```
import java.util.*;
public class GetListTest {
    /** Returns a list of integers from the keyboard. */
    public static ArrayList<Integer> getList() {
        ArrayList<Integer> a = new ArrayList<Integer>();
        < code to read integers into a>
        return a;
    }
    /** Write contents of ArrayList a. */
    public static void writeList(ArrayList<Integer> a) {
        System.out.println("List is : II + a);
    }
    public static void main(String[] args) {
        ArrayList<Integer> list = getList();
        wri teList (list) ;
    }
}
```

NOTE

1. The calls to `writeList(list)` and `getList()` do not need to be preceded by **GetListTest** plus a dot because main is not a client program: It is in the same class as `getList` and `writeList`.
2. If you omit the keyword static from the `getList` or `writeList` header, you get an error message like the following:

```
Can't make static reference to method
getList() in class GetListTest
```

The compiler has recognized that there was no object variable preceding the method call, which means that the methods were static and should have been declared as such.

Method Overloading

Overloaded methods are two or more methods in the same class (or a subclass of that class) that have the same name but different parameter lists. For example,

```
public class DoOperations {
  public int product(int n) { return n * n; }
  public double product(double x) { return x * x; }
  public double product(int x, int y) { return x * y; }
```

Method Overloading

- The compiler figures out which method to call by examining the method's signature. The signature of a method consists of the method's name and a list of the parameter types. Thus, the signatures of the overloaded product methods are

```
product(int)
product(double)
product(int, int)
```

- Note that for overloading purposes, the return type of the method is irrelevant. You can't have two methods with identical signatures but different return types. The compiler will complain that the method call is ambiguous.
- Having more than one constructor in the same class is an example of overloading. Overloaded constructors provide a choice of ways to initialize objects of the class.

Scope

- The scope of a variable or method is the region in which that variable or method is visible and can be accessed.
- The instance variables, static variables, and methods of a class belong to that class's scope, which extends from the opening brace to the closing brace of the class definition. Within the class all instance variables and methods are accessible and can be referred to simply by name (no dot operator!).
- A local variable is defined inside a method. It can even be defined inside a statement. Its scope extends from the point where it is declared to the end of the block in which its declaration occurs. A block is a piece of code enclosed in a {} pair. When a block is exited, the memory for a local variable is automatically recycled.

Scope

- Local variables take precedence over instance variables with the same name. (Using the same name, however, creates ambiguity for the programmer, leading to errors. You should avoid the practice.)

The **this** Keyword

- An instance method is always called for a particular object. This object is an implicit parameter for the method and is referred to with the keyword this. You are expected to know this vocabulary for the exam.
- In the implementation of instance methods, all instance variables can be written with the prefix this followed by the dot operator .

Example 1

- In the method call

```
obj.doSomething("Mary" ,num) ,
```

- where `obj` is some class object and `doSomething` is a method of that class, `"Mary"` and `num`, the parameters in parentheses, are explicit parameters, whereas obj is an implicit parameter.

Example 2

Here's an example where this is used as a parameter:

```
public class Person {
   private String name;
   private int age;
   public Person(String aName, int anAge) {
       name = aName;
       age = anAge;
   }
   /** Returns the String form of this person. */
   public String toString() { return name + " " + age; }
   public void printPerson() { System.out.println(this); }
   //Other variables and methods are not shown.
}
```

Example 2

- Suppose a client class has these lines of code:

```
Person p = new Person("Dan", 10);
p.printPerson();
```

- The statement

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

in the `printPerson` method means "print the current Person object." The output should be Dan 10. Note that System.out.println invokes the toString method of the Person class.

Example 3

- The `deposit` method of the `BankAccount` class can refer to balance as follows:

```
public void deposit(String acctPassword, double amount) {
    this.balance += amount;
}
```

- The use of this is unnecessary in the above example .

Example 4

- Consider a rational number class called `Rational`, which has two private instance variables:

```
private int num;      //numerator
private int denom;    //denominator
```

- Now consider a constructor for the Rational class:

```
public Rational(int num, int denom) {
   this.num = num;
   this.denom = denom;
}
```

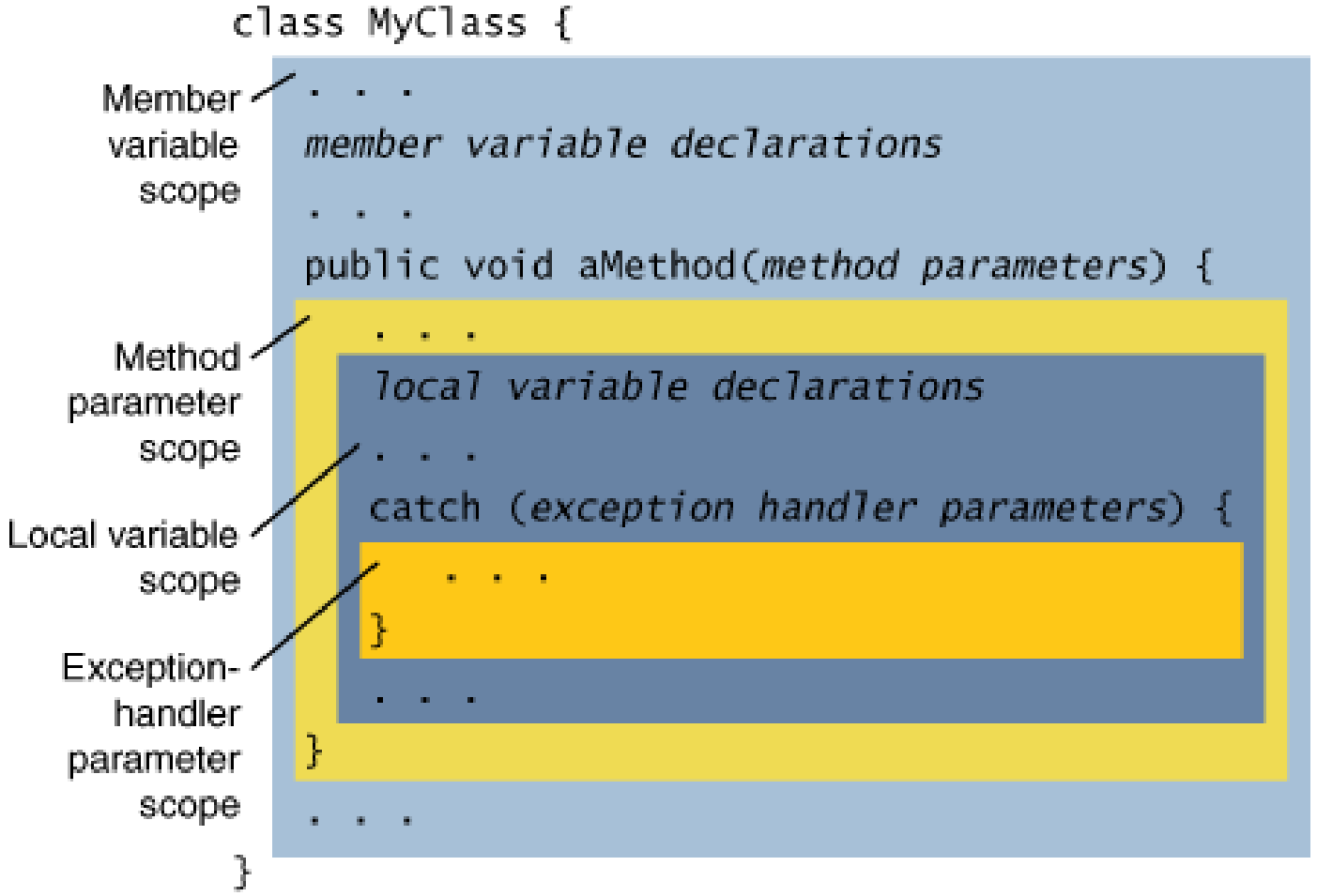
Example 4

- It is definitely not a good idea to use the same name for the explicit parameters and the private instance variables. But if you do, you can avoid errors by referring to this. num and `this.denom` for the current object that is being constructed. (This particular use of this will not be tested on the exam.)

SECTION 3

Properties of Classes

Local Variable Versus Global Variable



Global Variables:

Member Properties:

Instance Variable - double radius;

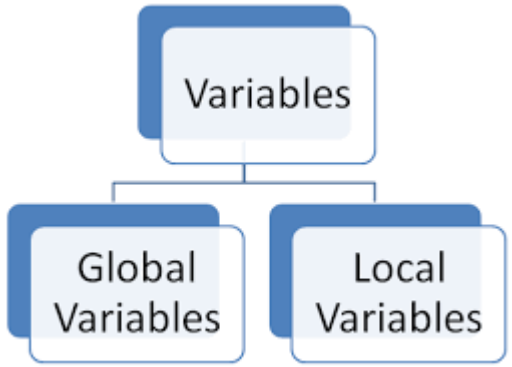
Class Variable (static) – static int num;

Local Variables:

Arguments

Local Variables at Method level

Block level local Variables

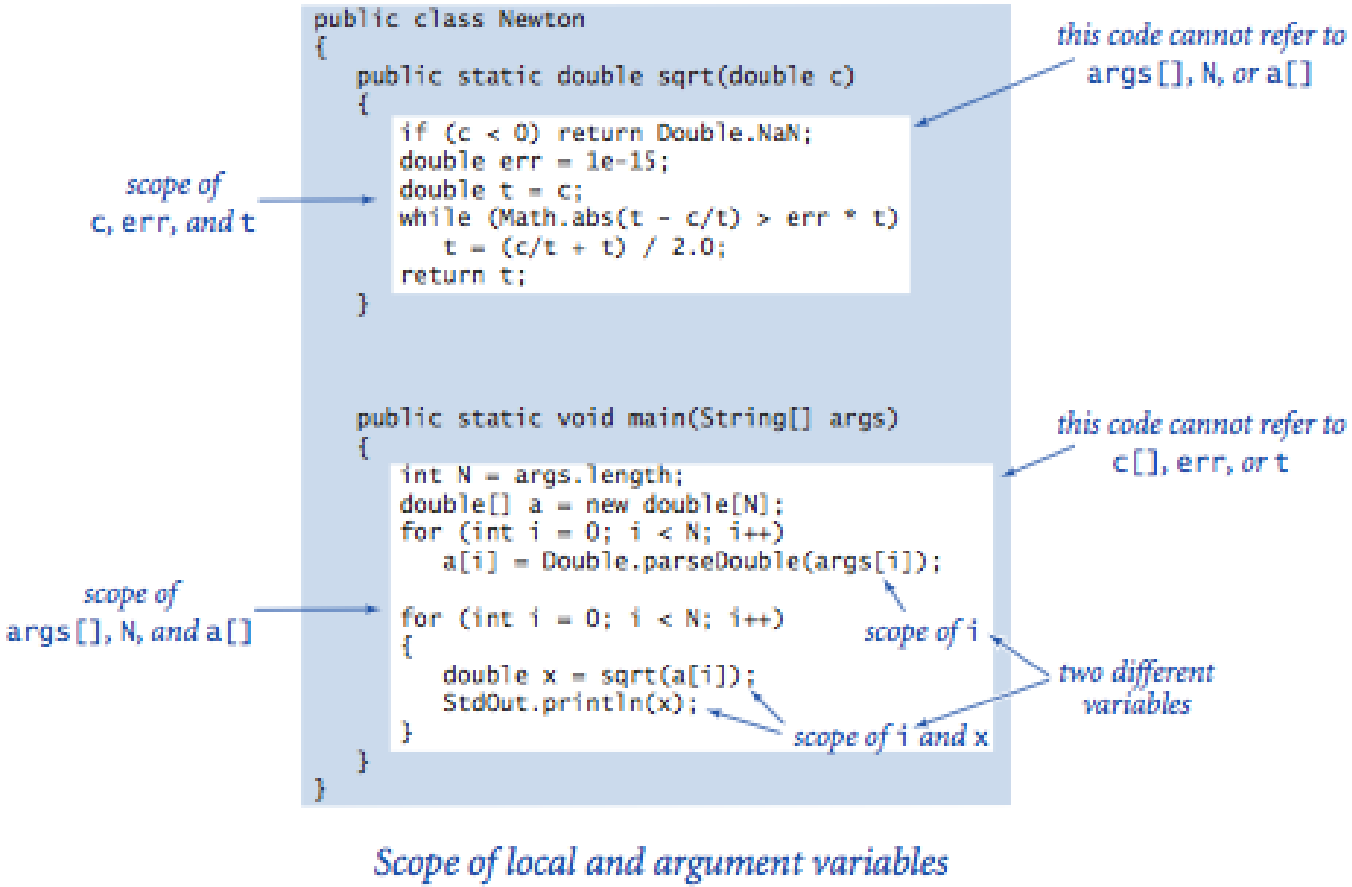


Summary of Local Variables

variable	*purpose*	*example*	*scope*
instance	data-type value	rx	class
parameter	pass value from client to method	x	method
local	temporary use within method	dx	block

Static Methods

(Program control structure Only, not related to data)



Scope of local and argument variables

The use of static methods is easy to understand. For example, when you write Math.abs(a-b) in a program, the effect is as if you were to replace that code by the value that is computed by Java's Math.abs() method when presented with the value a-b.

If you think about what the computer has to do to create this effect, you will realize that it involves changing a program's **flow of control**.

Data and Methods Visibility

Modifier on Members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass in a different package	Accessed from a different package
public	O	O	O	O
protected	O	O	O	X
default	O	O	X	X
private	O	X	X	X

```
package p1;

public class C1 {
  public int x;
  int y;
  private int z;

  public void m1() {
  }
  void m2() {
  }
  private void m3() {
  }
}
```

```
public class C2 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    can access o.y;
    cannot access o.z;

    can invoke o.m1();
    can invoke o.m2();
    cannot invoke o.m3();
  }
}
```

```
package p2;

public class C3 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    cannot access o.y;
    cannot access o.z;

    can invoke o.m1();
    cannot invoke o.m2();
    cannot invoke o.m3();
  }
}
```

```
package p1;

class C1 {
  ...
}
```

```
public class C2 {
  can access C1
}
```

```
package p2;

public class C3 {
  cannot access C1;
  can access C2;
}
```

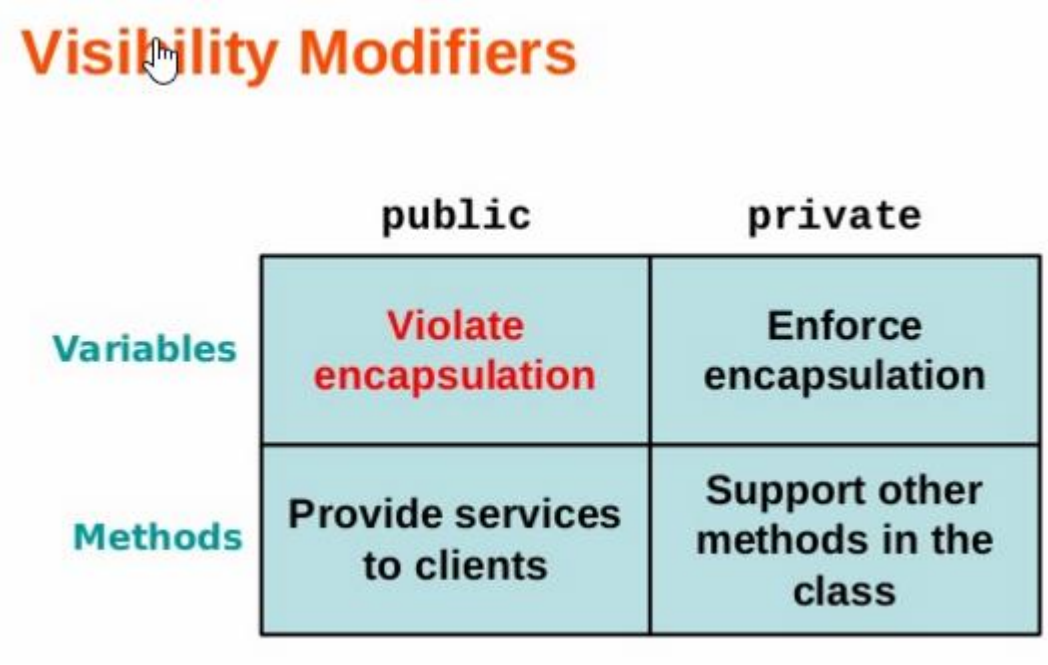
The private modifier restricts access to within a class, the default modifier restricts access to within a package, and the public modifier enables unrestricted access.

Classes and Objects Design Styles

(Classification by Styles)

Immutable Classes (Object): All Private Data Fields, No Mutators and No Accessor Method Returning Reference Data.

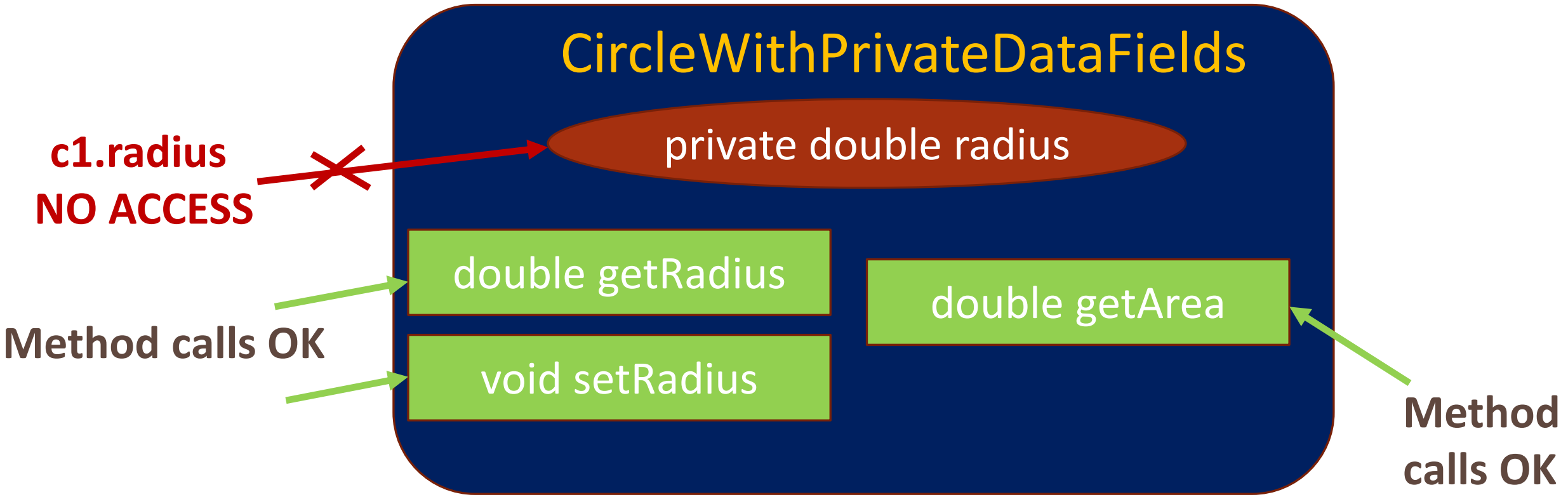
Classes and Objects (4)



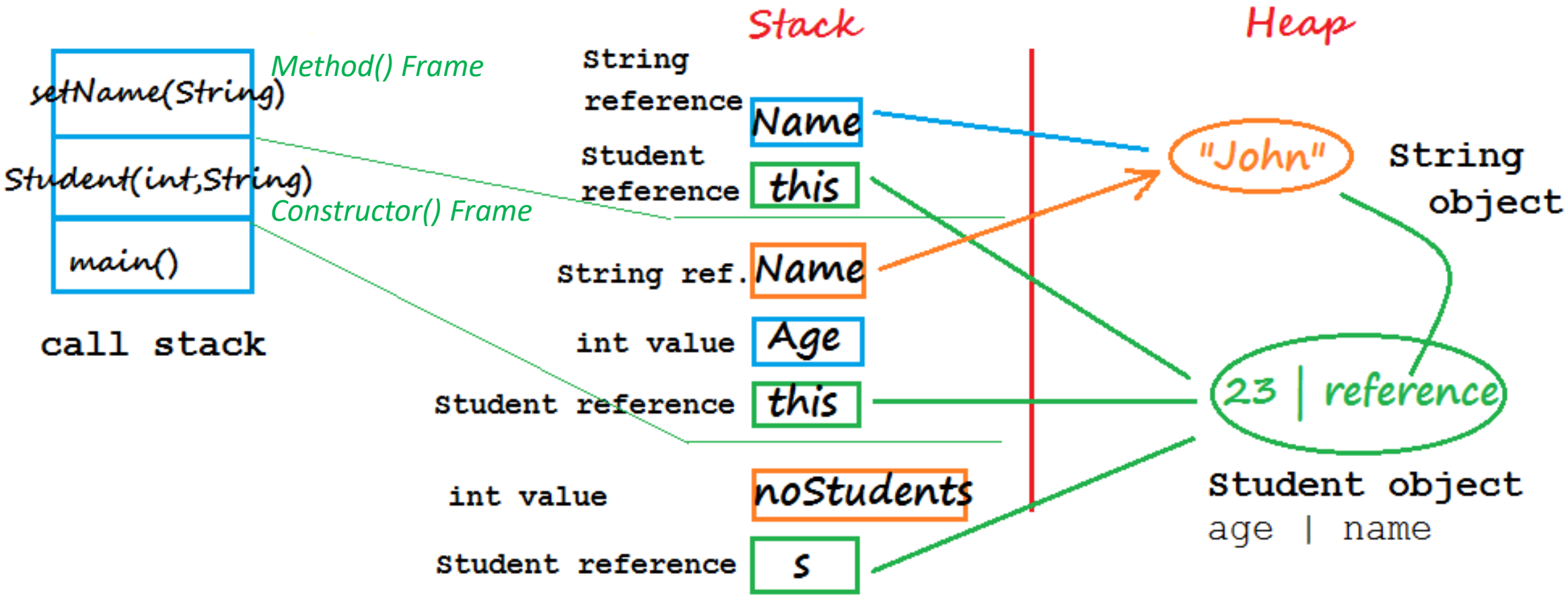
Lifetime and Scope

- Static variable get life as soon as class is loaded into JVM and is available till class is removed from JVM or JVM is shutdown.
- And its scope is class scope means it is accessible throughout the class.

Encapsulation Using Private Data Fields

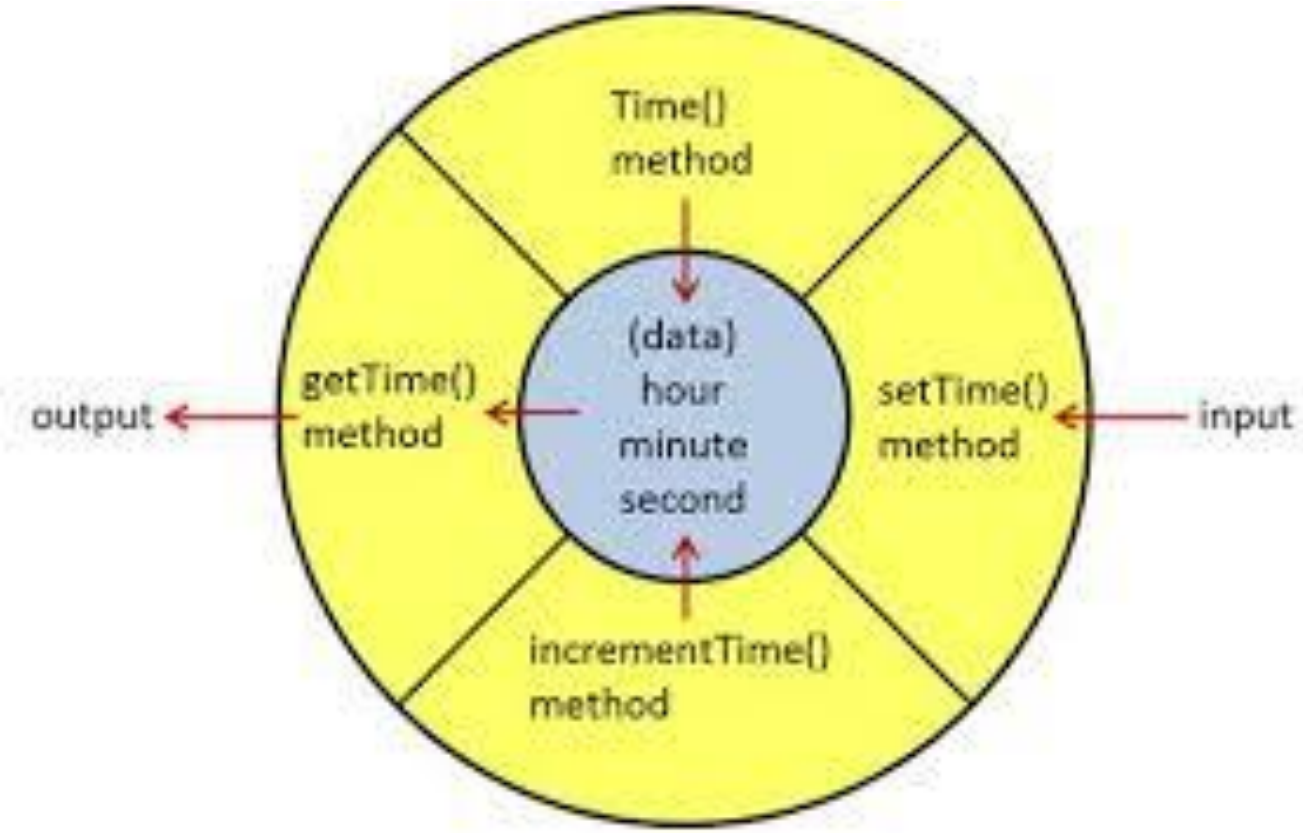


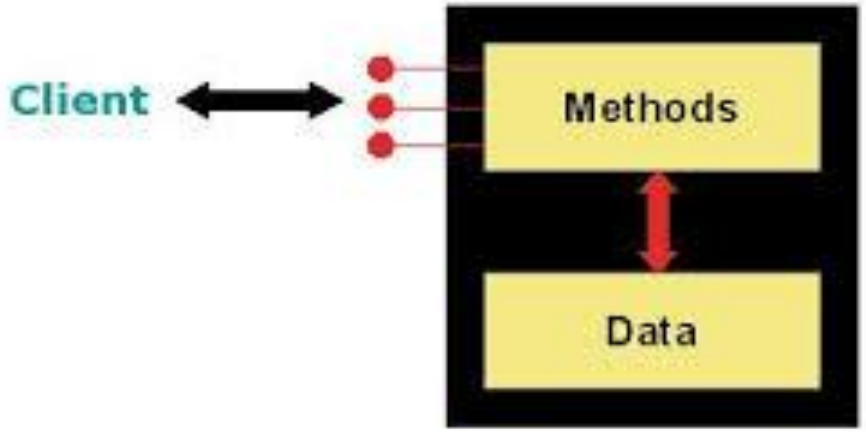
Memory Allocation



First Way: Accessor/Mutator Methods

(For Encapsulated Objects)





SECTION 4

References

References

- **Reference vs. Primitive Data Types**
- Simple built-in data types, like double and int, as well as types char and boolean, are primitive data types. All objects and arrays are reference data types, such as String, Random, int [] , String[][], Cat (assuming there is a Cat class in the program), and so on. The difference between primitive and reference data types lies in the way they are stored.
- Consider the statements

```
int num1 = 3;
int num2 = num1;
```

References

- The variables `num1` and `num2` can be thought of as memory slots, labeled num1 and num2, respectively:

num1 num2

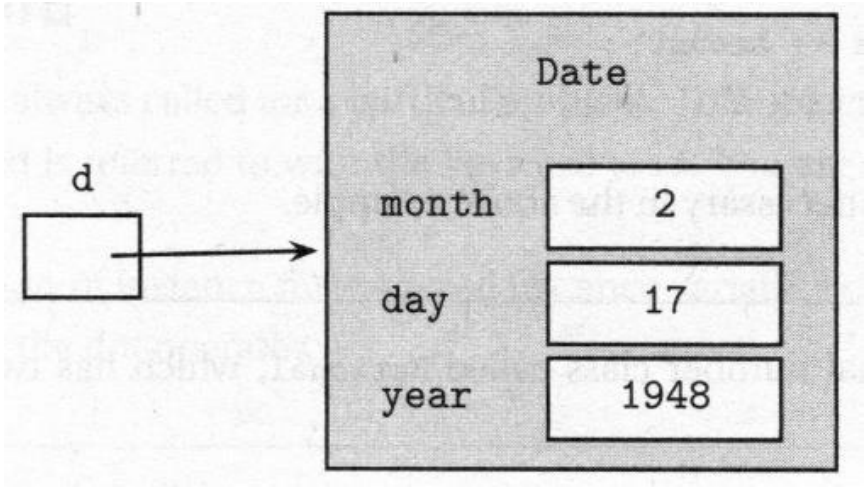
3 3

- If either of the above variables is now changed, the other is not affected. Each has its own memory slot.
- Contrast this with the declaration of a reference data type. Recall that an object is created using new:

```
Date d = new Date(2, 17, 1948);
```

References

- This declaration creates a reference variable d that refers to a Date object. The value of d is the address in memory of that object:

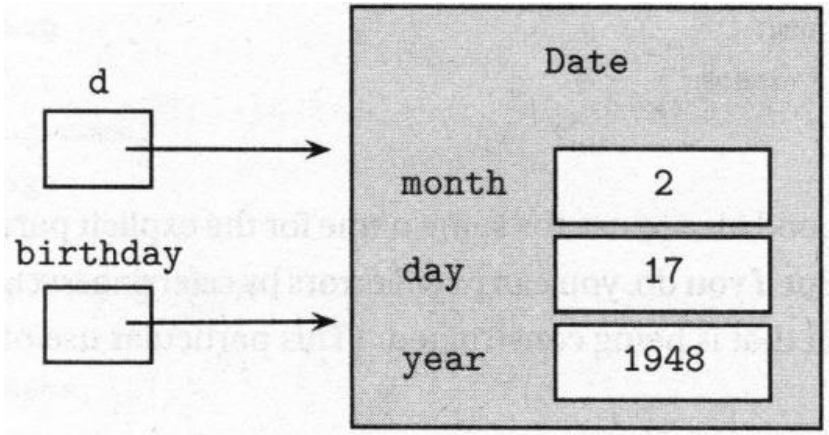


- Suppose the following declaration is now made:

```
Date birthday = d;
```

References

- This statement creates the reference variable birthday, which contains the same address as d:



- Having two references for the same object is known as aliasing. Aliasing can cause unintended problems for the programmer. The statement will automatically change the object referred to by birthday as well.

References

- What the programmer probably intended was to create a second object called birthday whose attributes exactly matched those of `d`. This cannot be accomplished without using new.
- For example,

```
Date birthday = new Date(d.getMonth(),
                         d.getDay(),
                         d.getYear());
```

- The statement `d.changeDate()` will now leave the birthday object unchanged.

The Null Reference

- The declaration

```
BankAccount b;
```

defines a reference b that is uninitialized. (To construct the object that b refers to requires the new operator and a `BankAccount` constructor.) An uninitialized object variable is called a null reference or null pointer. You can test whether a variable refers to an object or is uninitialized by using the keyword **`null`**:

```
if (b == null)
```

- If a reference is not null, it can be set to null with the statement

```
b = null;
```

The Null Reference

- An attempt to invoke an instance method with a null reference may cause your program to terminate with a **NullPointerException**. For example,

```
public class PersonalFinances {.
BankAccount b;
// b is a null reference
b.withdraw(acctPassword, amt);
// throws a NullPointerException if b not constructed with new
```

The Null Reference

NOTE

- If you fail to initialize a local variable in a method before you use it, you will get a compile-time error. If you make the same mistake with an instance variable of a class, the compiler provides reasonable default values for primitive variables (0 for numbers, false for booleans), and the code may run without error.
- However, if you don't initialize reference instance variables in a class, as in the above example, the compiler will set them to null. Any method call for an object of the class that tries to access the null reference will cause a run-time error: The program will terminate with a **NullPointerException**.

SECTION 5

Method Parameters

Formal VS. Actual Parameters

- The header of a method defines the parameters of that method. For example, consider the withdraw method of the `BankAccount` class:

```
public class BankAccount {
public void withdraw(String acctPassword, double amount)
```

- This method has two explicit parameters, acctPassword and amount. These are dummy or formal parameters. Think of them as placeholders for the pair of actual parameters or arguments that will be supplied by a particular method call in a client program.

Formal VS. Actual Parameters

- For example,

```
BankAccount b = new BankAccount("TimB", 1000);
b.withdraw("TimB", 250);
```

- Here "TimB" and 250 are the actual parameters that match up with acctPassword and amount for the wi thdraw method.

Formal VS. Actual Parameters

NOTE

1. The number of arguments in the method call must equal the number of parameters in the method header, and the type of each argument must be compatible with the type of each corresponding parameter.
2. In addition to its explicit parameters, the wi thdraw method has an implicit parameter, this, the BankAccount from which money will be withdrawn. In the method call

   ```
   b.withdraw("TimB", 250);
   ```

   the actual parameter that matches up with this is the object reference b.

Passing Primitive Types as Passing Parameters

- Parameters are passed by value. For primitive types this means that when a method is called, a new memory slot is allocated for each parameter. The value of each argument is copied into the newly created memory slot corresponding to each parameter.
- During execution of the method, the parameters are local to that method. Any changes made to the parameters will not affect the values of the arguments in the calling program. When the method is exited, the local memory slots for the parameters are erased.

Passing Primitive Types as Passing Parameters

- Here's an example. What will the output be?

```
public class ParamTest {
  public static void foo(int x, double y){
      x = 3;
      y = 2.5;
  }
  public static void main(String[] args){
      int a = 7;
      double b = 6.5;
      foo(a, b);
      System.out.println(a + " " + b);
  }
}
```

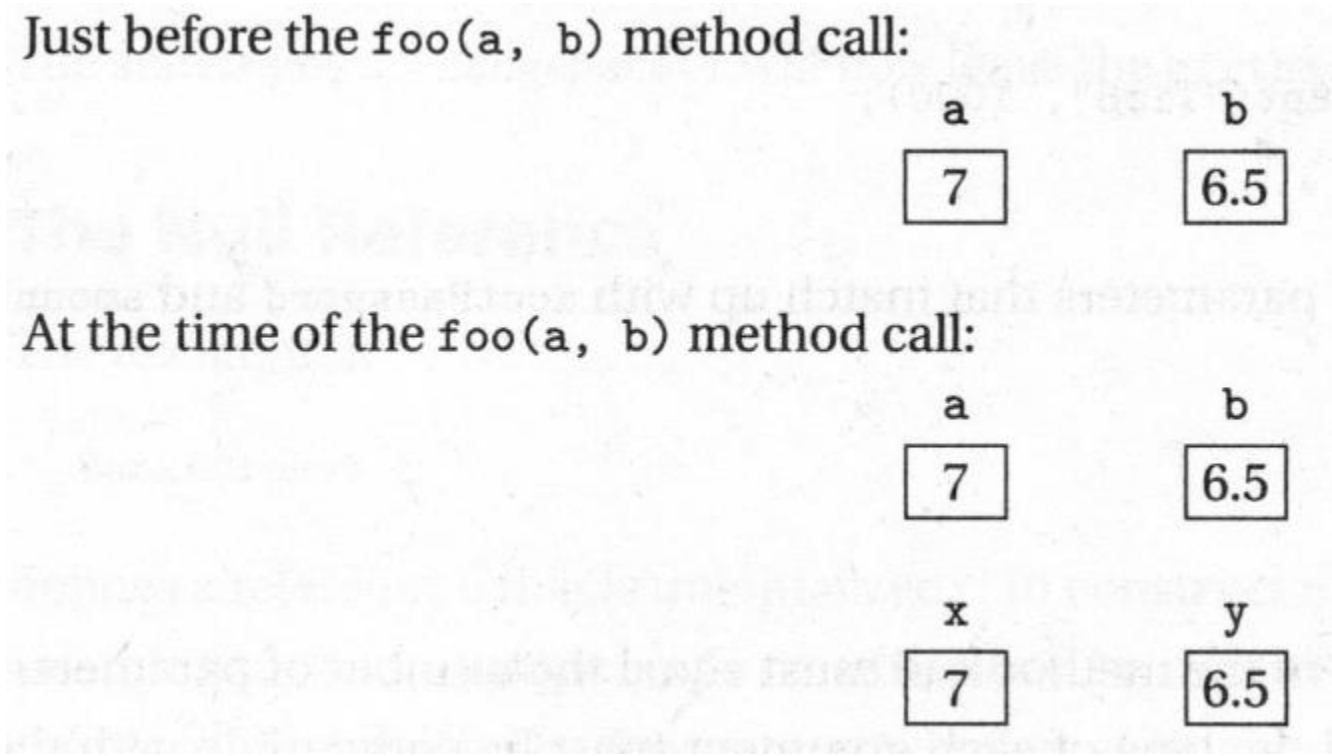
Passing Primitive Types as Passing Parameters

- The output will be

```
7  6.5
```

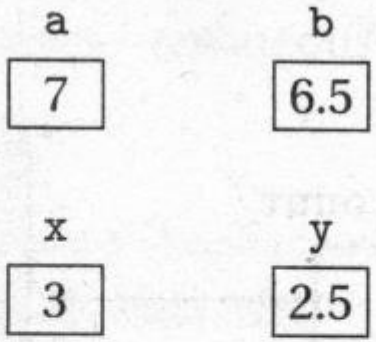
- The arguments a and b remain unchanged, despite the method call!
- This can be understood by picturing the state of the memory slots during execution of the program.

Passing Primitive Types as Passing Parameters



Passing Primitive Types as Passing Parameters

Just before exiting the method: (Note that the values of x and y have been changed.)



After exiting the method: (Note that the memory slots for x and y have been reclaimed. The values of a and b remain unchanged.)

Passing Objects as Parameters

- In Java both primitive types and object references are passed by value. When an object's reference is a parameter, the same mechanism of copying into local memory is used. The key difference is that the address (reference) is copied, not the values of the individual instance variables.
- As with primitive types, changes made to the parameters will not change the values of the matching arguments. What this means in practice is that it is not possible for a method to replace an object with another one-you can't change the reference that was passed. It is, however, possible to change the state of the object to which the parameter refers through methods that act on the object .

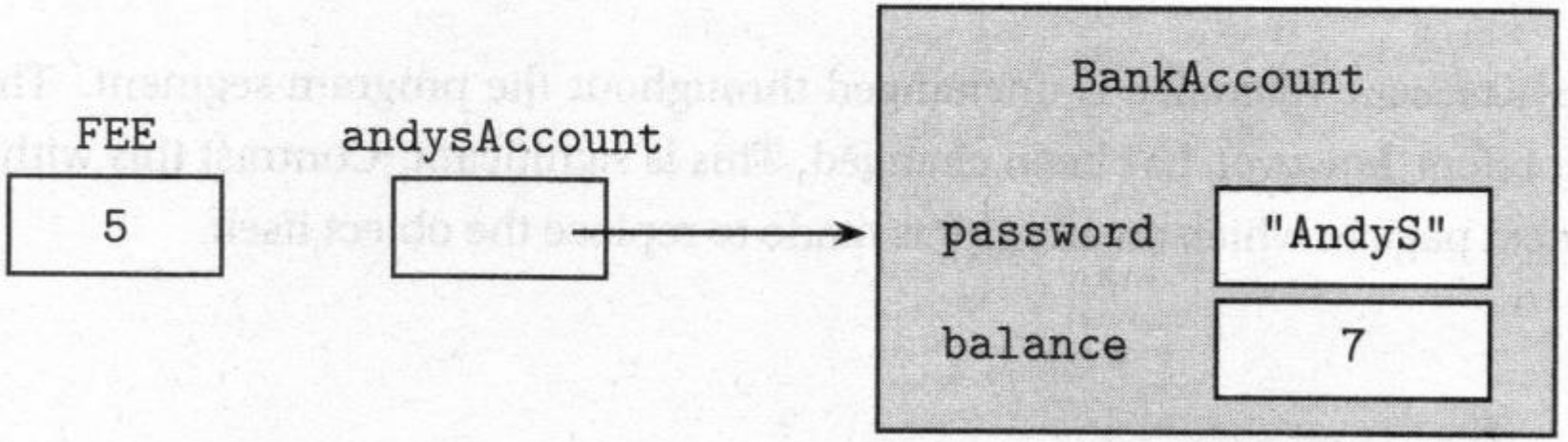
Example 1

/** Subtracts fee from balance in b if current balance too low. */

```
public static void chargeFee(BankAccount b,String password,double fee)
{
    final double MIN_BALANCE = 10.00;
    if (b.getBalance() < MIN_BALANCE)
    b.withdraw(password, fee);

}

public static void main(String[] args) {
    final double FEE = 5.00;
    BankAccount andysAccount = new BankAccount("AndyS", 7.00);
    chargeFee (andysAccount, "AndyS", FEE);

}
```

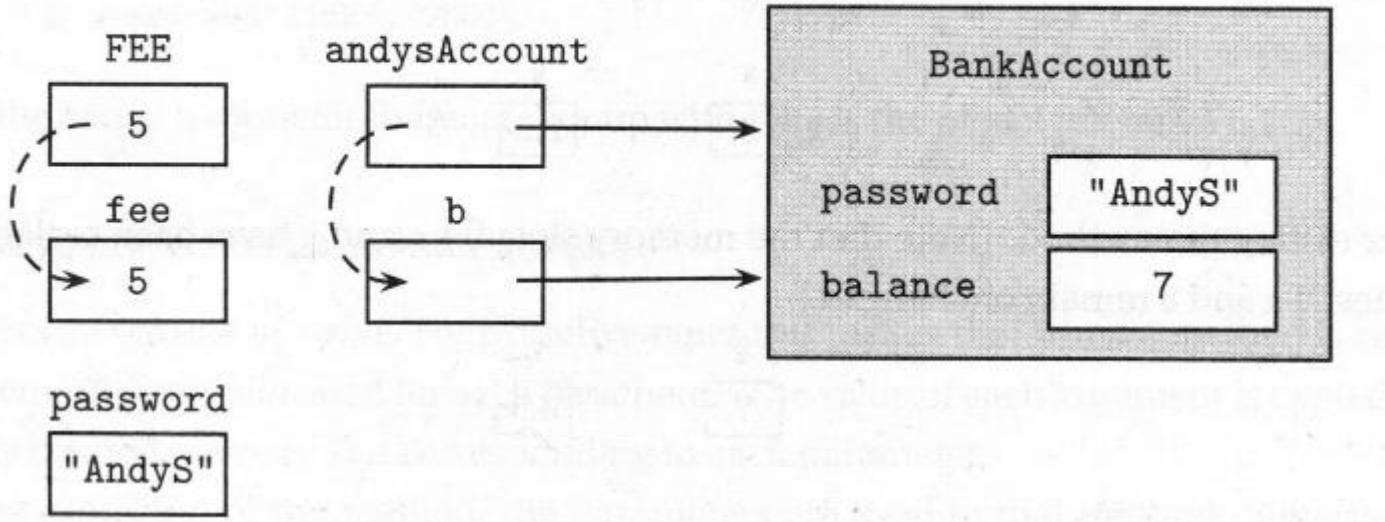
Example 1

Here are the memory slots before the `chargeFee` method call:



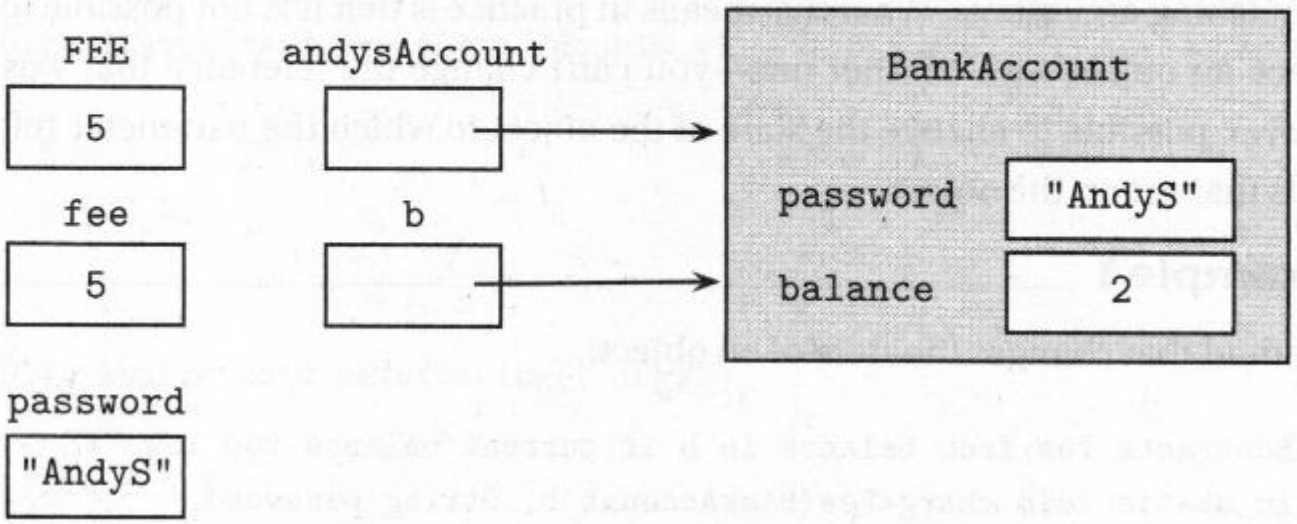
Example 1

At the time of the `chargeFee` method call, copies of the matching parameters are made:



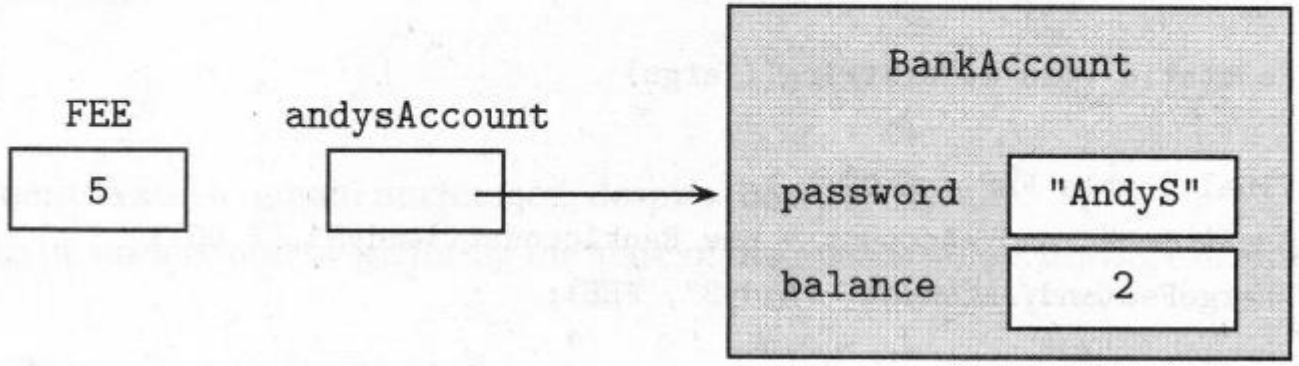
Example 1

Just before exiting the method: (The `balance` field of the `BankAccount` object has been changed.)



Example 1

After exiting the method: (All parameter memory slots have been erased, but the object remains altered.)



NOTE

The `andysAccount` reference is unchanged throughout the program segment. The object to which it refers, however, has been changed. This is significant. Contrast this with Example 2 on the next page in which an attempt is made to replace the object itself.

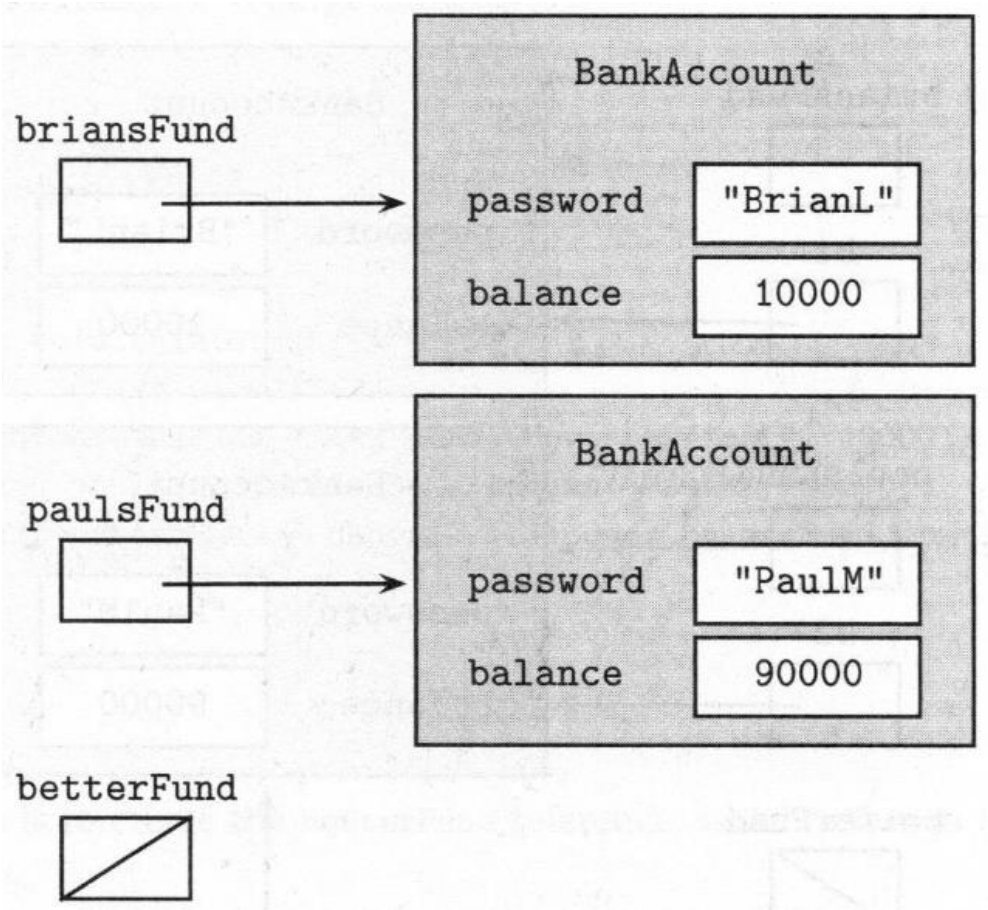
Example 2

- A chooseBestAccount method attempts-erroneously-to set its betterFund parameter to the BankAccount with the higher balance:

```
public static void chooseBestAccount(
      BankAccount better, BankAccount b1, BankAccount b2){
      if (b1.getBalance() > b2.getBalance())
            better b1;
      else
            better b2;
}
public static void main(String[] args) {
  BankAccount briansFund = new BankAccount("BrianL", 10000);
  BankAccount paulsFund = new BankAccount("PaulM", 90000);
  BankAccount betterFund = null;
  chooseBestAccount(betterFund, briansFund, paulsFund);
}
```

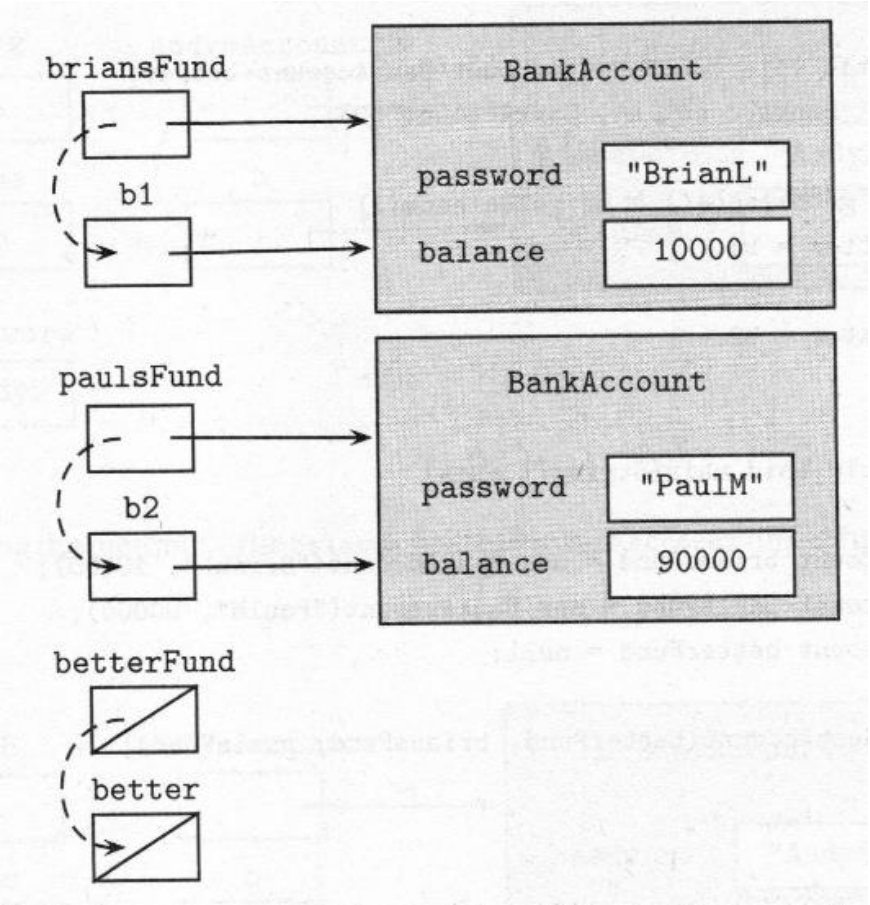
Example 2

- The intent is that better Fund will be a reference to the `paulsFund` object after execution of the `chooseBestAccount` statement. A look at the memory slots illustrates why this fails.
- Before the `chooseBestAccount` method call:



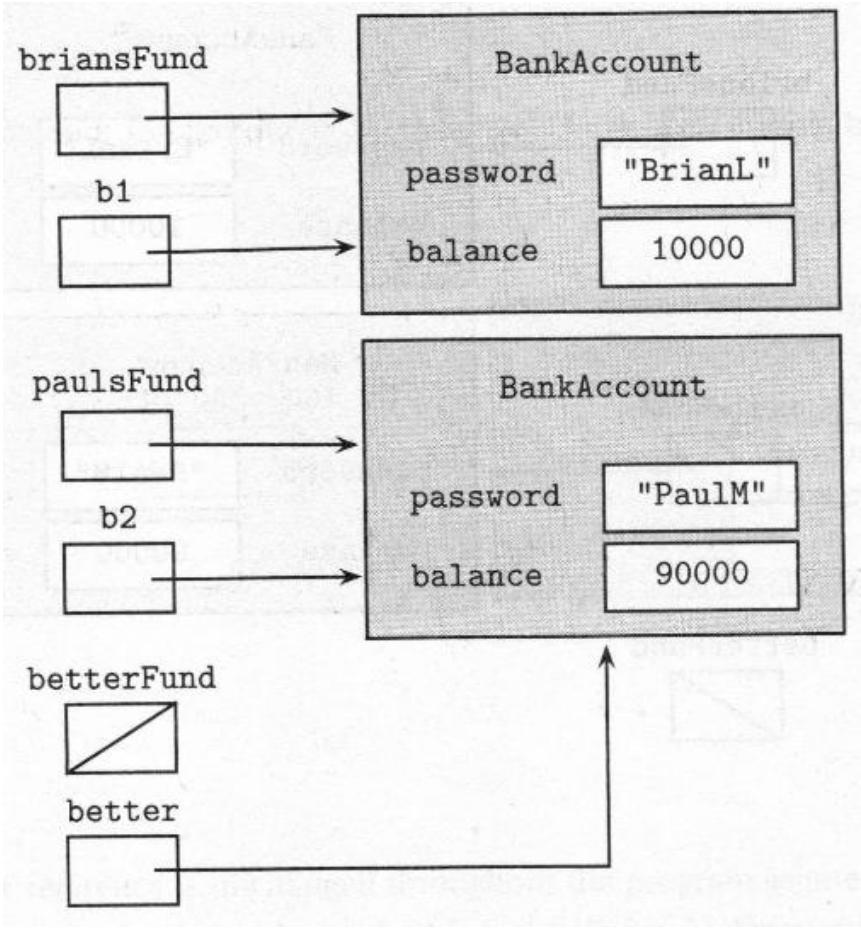
Example 2

- At the time of the `chooseBestAccount` method call, copies of the matching references are made:



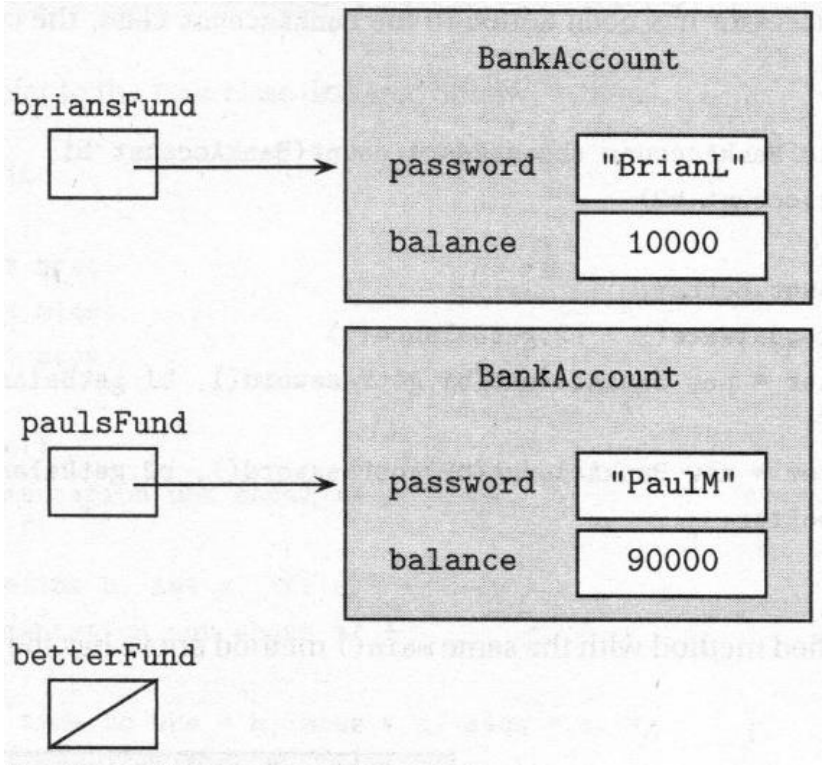
Example 2

- Just before exiting the method, the value of better has been changed; `betterFund`, however, remains unchanged:



Example 2

- After exiting the method, all parameter slots have been erased:



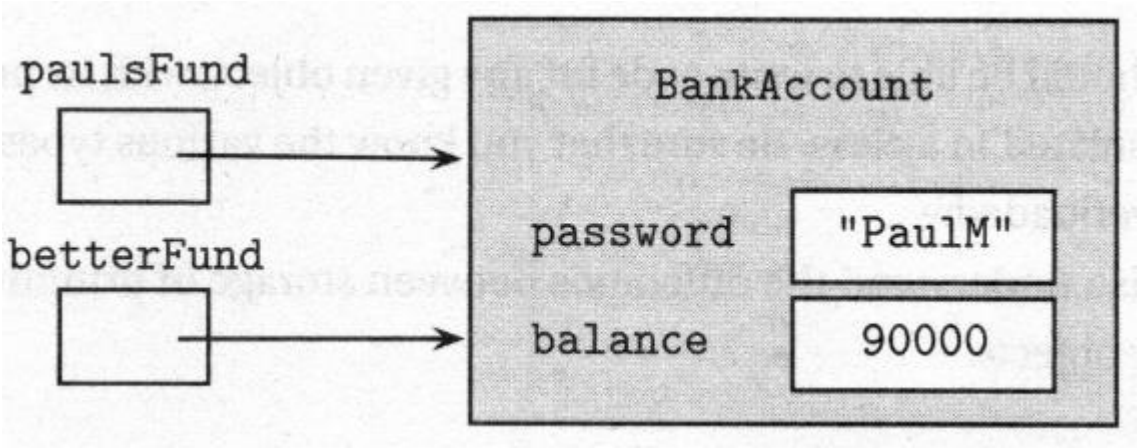
Example 2

- Note that the `betterFund` reference continues to be null, contrary to the programmer's intent. The way to fix the problem is to modify the method so that it returns the better account.
- Returning an object from a method means that you are returning the address of the object.

```
public static BankAccount chooseBestAccount(
    BankAccount bl, BankAccount b2)
    BankAccount better;
    if (bl.getBalance() > b2.getBalance())better = bl;
    else better = b2;
    return better;
}
public static void main(String[] args) {
    BankAccount briansFund = new BankAccount(lOBrianLlO, 10000);
    BankAccount paulsFund  = new BankAccount(lOPaulMlO, 90000);
    BankAccount betterFund = chooseBestAccount(briansFund, paulsFund);
}
```

NOTE

The effect of this is to create the better Fund reference, which refers to the same object as `paulsFund`:



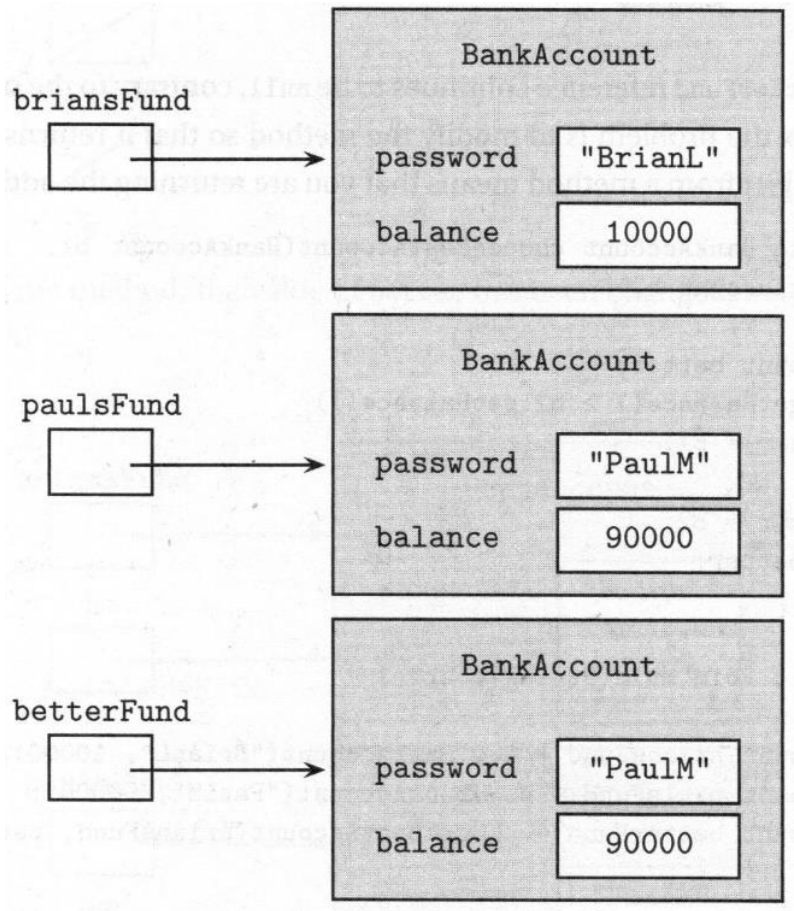
Example 2

- What the method does not do is create a new object to which betterFund refers. To do that would require the keyword new and use of a BankAccount constructor. Assuming that a getPassword() accessor has been added to the BankAccount class, the code would look like this:

```
public static BankAccount chooseBestAccount(BankAccount b1, BankAccount b2)
{
     BankAccount better;
     if (b1.getBalance() > b2.getBalance())
     better = new BankAccount(b1.getPassword(), b1.getBalance());
     better = new BankAccount(b2.getPassword(), b2.getBalance());
     return better;

}
```

Example 2

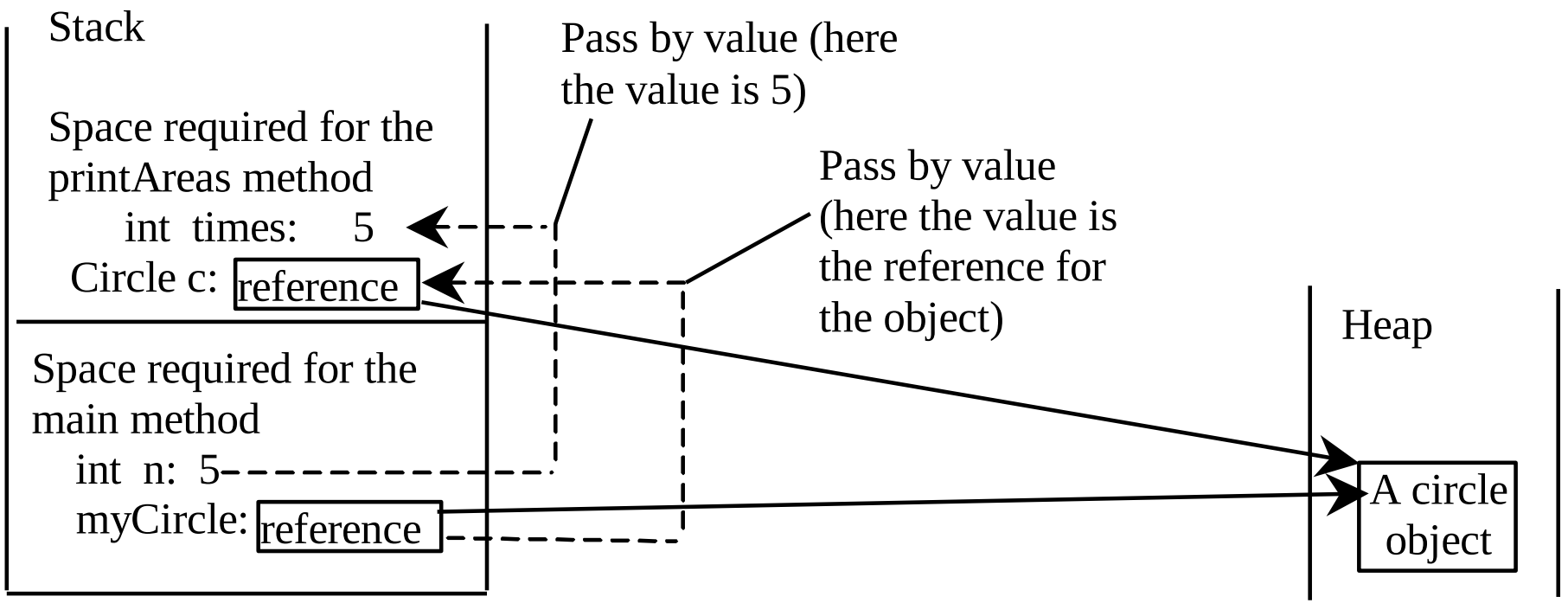
- Using this modified method with the same main 0 method above has the following effect:
- Modifying more than one object in a method can be accomplished using a wrapper class (see p. 173).



Passing Objects to Methods

- Passing by value for primitive type value (the value is passed to the parameter)
- Passing by value for reference type value (the value is the reference to the object)

Passing Objects to Methods, cont.



Immutable Objects and Classes

You can define immutable classes to create immutable objects. The contents of immutable objects cannot be changed.

The String class is immutable.

If a class is immutable, then all its data fields must be private and it cannot contain public setter methods for any data fields. A Class with **all private data fields** and **no mutators** is not necessarily immutable. For example, the following Student class has all private data fields and no setter methods,

Immutable Objects and Classes

- For a class to be immutable, it must meet the following requirements:
- All **data fields** must be **private**.
- There can't be any mutator methods for data fields. (**No Mutator**)
- No accessor methods can return a reference to data field that is mutable. (**No accessor method for reference data.)**

SECTION 6

Summary

Summary

- By now you should be able to write code for any given object, with its private data fields and methods encapsulated in a class. Be sure that you know the various types of methods-static, instance, and overloaded.
- You should also understand the difference between storage of primitive types and the references used for objects.