Unit objectives

- After completing this unit, you should be able to:
  - Declare a Java class
  - Define constructors
  - Create methods and fields, and set the appropriate modifier
  - Explain how memory is managed in Java
  - Outline the role of packages in Java

Classes

- Encapsulate attributes (fields) and behavior (methods)
  - Attributes and behavior are members of the class
- Members may belong to either of the following:
  - The whole class (class variables and methods, indicated by the keyword static)
  - Individual objects (instance variables and methods)
- Classes can be:
  - Independent of each other
  - Related by inheritance (superclass/subclass)
  - Related by type (interface)

Implementing classes

- Classes are grouped into packages
  - A package contains a collection of logically-related classes
- Source code files have the extension .java
  - There is one public class per .java file
- A class is like a blueprint; a class is used to create an object or instance of the class typically

Class declaration

- A class declaration specifies a type
  - The identifier in a class declaration specifies the name of the class
  - The optional extends clause indicates the superclass
  - The optional implements clause lists the names of all of the interfaces that the class implements

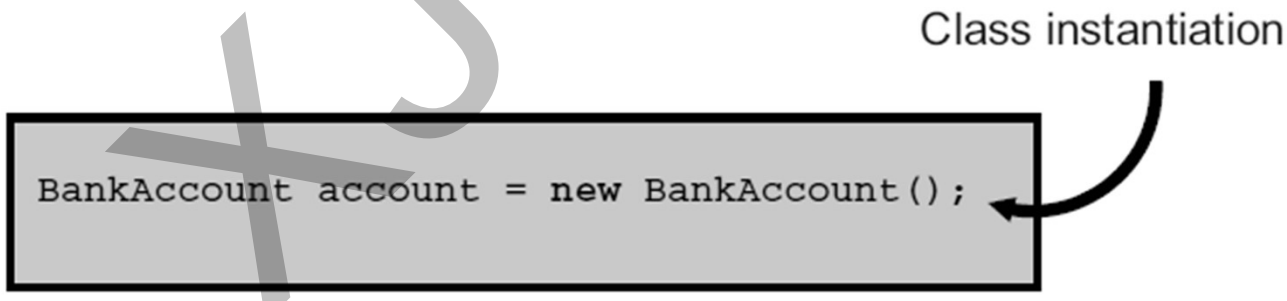
```
public class BankAccount extends Account
    implements Serializable, BankStuff {

    // Class Body
}
```

Class modifiers

- The declaration may include class modifiers (public,abstract, final) which affect how the class can be used
  - If the class is declared public, it may be accessed by any Java code that can access its containing package
    - Otherwise it may be accessed only from within its containing package
  - Abstract classes can contain anything that a normal class can contain (variables, methods, constructors)
    - Cannot be instantiated, only subclassed
    - Provide common information for subclasses
  - A class is declared final if it permits no subclasses

Constructors

- The class body contains at least one constructor, which is a method that sets up a new instance of a class
  - The method has the same name as the class
- Use the new keyword with a constructor to create instances of a class



Memory management in Java

- Since Java does not use pointers, memory addresses cannot be accidentally or maliciously overwritten
- The problems inherent in user allocated and deallocated memory are avoided, since the Java Virtual Machine handles all memory management
  - Programmers do not have to keep track of the memory they allocate from the heap and explicitly deallocate it

More about constructors

- **Constructors are used to create and initialize objects**
  - **A constructor always has the same name as the class it constructs (case-sensitive)**
- **Constructors have no return type**
  - **Constructors return no value, but when used with new return a reference to the new object**

```
public BankAccount(String name) {
   setOwner(name);
}
```

Constructor definition

Constructor use

```
BankAccount account = new BankAccount("Joe Smith");
```

Default constructors

- The constructor with no arguments is a default constructor
- The Java platform provides a default constructor only if you do not explicitly define any constructor
- When defining a constructor, you should also provide a default constructor

Overloading constructors

- There may be any number of constructors with different parameters
  - This is called overloading
- Constructors are commonly overloaded to allow for different ways of initializing instances

```
BankAccount new_account =
   new BankAccount();

BankAccount known_account =
   new BankAccount(account_number);

BankAccount named_account =
   new BankAccount("My Checking Account");
```

Constructor example

- In a constructor, the keyword this is used to refer to other constructors in the same class

```
...

public BankAccount(String name) {
    super();
    owner = name;
}

public BankAccount() {
    this("TestName");
}

public BankAccount(String name, double initialBalance) {
    this(name);
    setBalance(initialBalance);
}
...
```

Constructor chaining

- Superclass objects are built before the subclass
  - super(argument list) initializes superclass members
- The first line of your constructor can be one of:
  - super(argument list);
  - this(argument list);
- You cannot use both super() and this() in the same constructor
- The compiler supplies an implicit super() constructor for all constructors

Java destructors?

- Java does not have the concept of a destructor for objects that are no longer in use
- Deallocation of memory is done automatically by the JVM
  - A background process called the garbage collector reclaims the memory of unreferenced objects
  - The association between an object and an object reference is severed by assigning another value to the object reference, for example:

```
objectReference = null;
```

  - An object with no references is a candidate for deallocation during garbage collection

Garbage collector

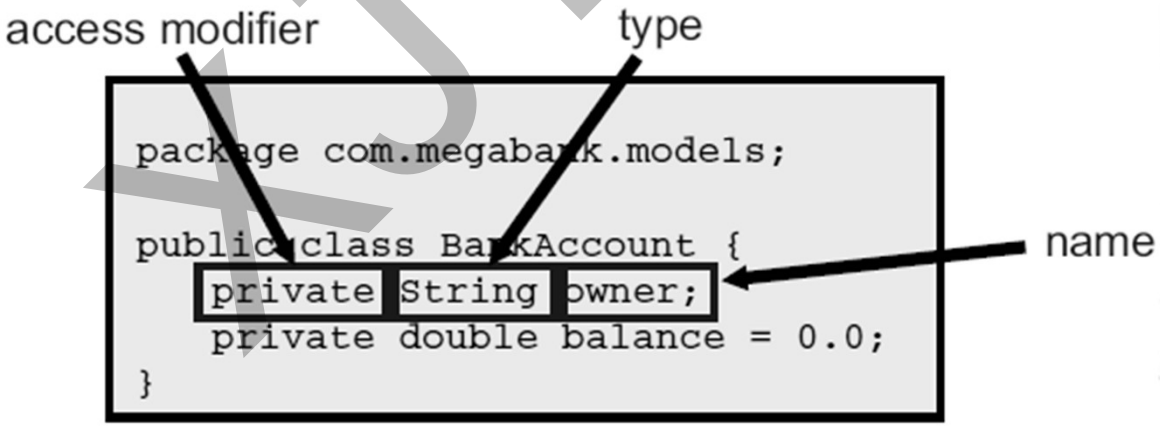
- The garbage collector sweeps through the JVM's list of objects periodically and reclaims the resources held by unreferenced objects
- All objects that have no object references are eligible for garbage collection
  - References out of scope, objects to which you have assigned null, and so forth
- The JVM decides when the garbage collector is run
  - Typically, the garbage collector is run when memory is low
  - May not be run at all
  - Unpredictable timing

Working with the garbage collector

- You cannot prevent the garbage collector from running, but you can request it to run soon
  - System.gc();
  - This is only a request, not a guarantee
- The finalize() method of an object will be run immediately before garbage collection occurs
  - This method should only be used for special cases (such as cleaning up memory allocation from native calls) because of the unpredictability of the garbage collector
  - Things like open sockets, files, and so forth should be cleaned up during normal program flow before the object is dereferenced

Fields

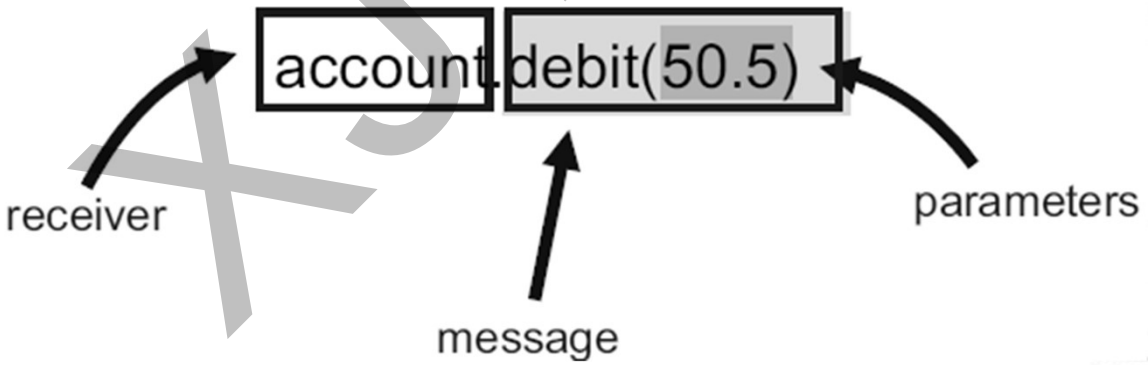
- Objects retain state in fields
  - Fields are defined as part of the class definition
  - Each instance gets its own copy of the instance variables
- Fields can be initialized (if desired) when declared
  - Default values will be used if fields are not initialized



Messages

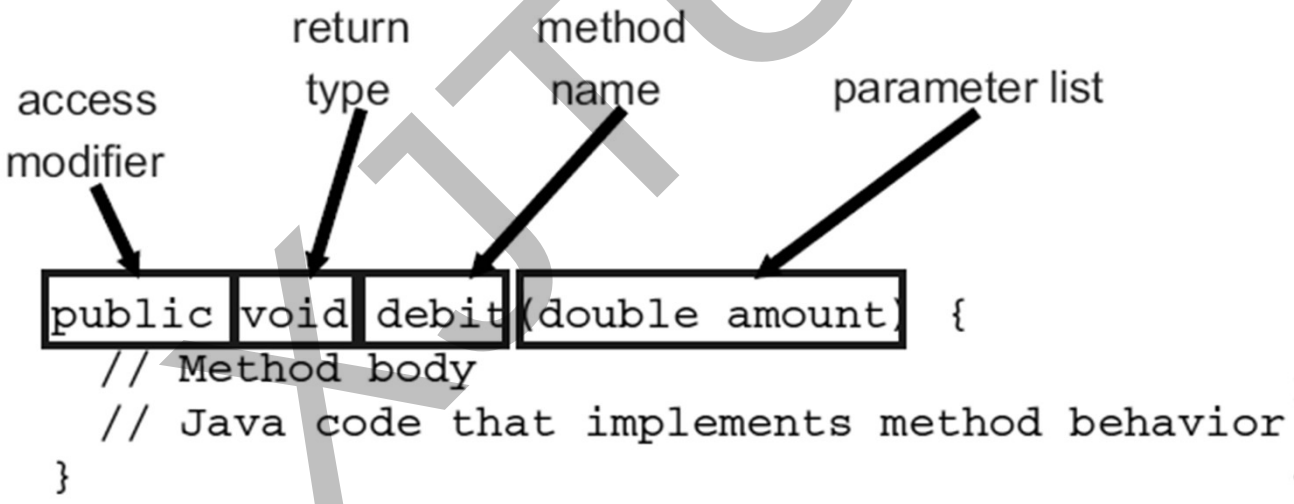
- Use messages to invoke the behavior of an object

```
BankAccount account = new BankAccount();
account.setOwner("Smith");
account.credit(1000.0);
account.debit(50.5);
...
```



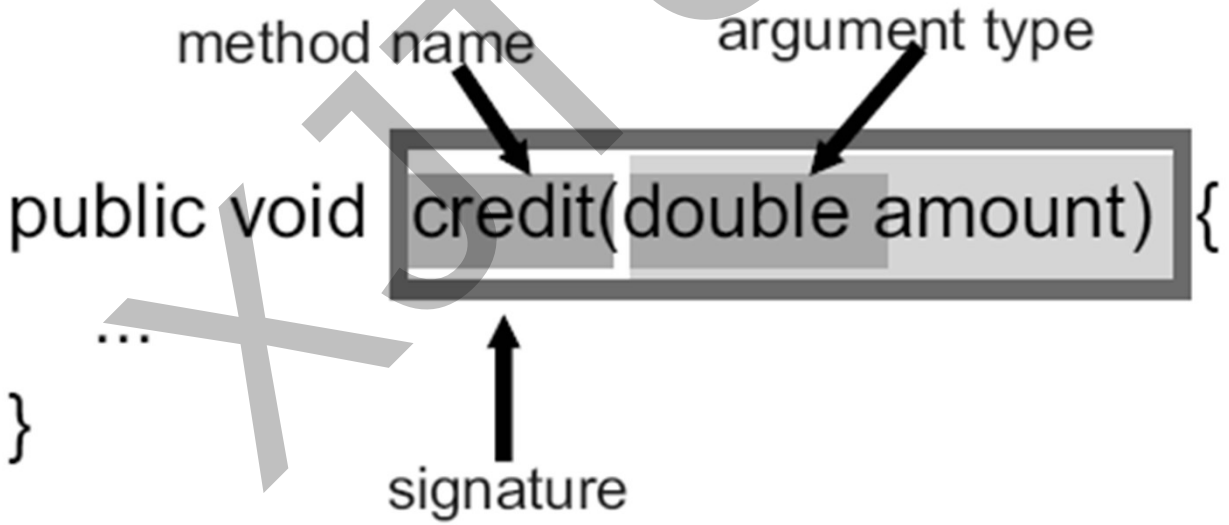
Methods

- Methods define how an object responds to messages
- Methods define the behavior of the class
  - All methods belong to a class



Method signatures

- A class can have many methods with the same name
  - Each method must have a different signature
- The method signature consists of:
  - The method name
  - Argument number and types



Method parameters

- Arguments (parameters) are passed:
  - By value for primitive types
  - By object reference for reference types
- Primitive values cannot be modified when passed as an argument

```
public void method1(){
   int a = 0;
   System.out.println(a); // outputs 0
   method2(a);
   System.out.println(a); // outputs 0
}
```

```
void method2(int a){
   a = a + 1;
}
```

Returning from methods

- Methods return at most one value or one object
  - If the return type is void, the return statement is optional
- The return keyword is used to return control to the calling method
  - There may be several return statements in a method; the first one reached will be executed

```
public void debit(double amount) {
    if (amount > getBalance()) return;
    setBalance(getBalance() - amount);
}
```

```
public String getFullName() {
   return getFirstName() + " " + getLastName();
}
```

Invoking methods

- To call a method, use the dot ñ.ò operator
  - The same operator is used to call both class and instance methods
  - If the call is to a method of the same class, the dot operator is not necessary

```
BankAccount account = new BankAccount();
account.setOwner("Smith");
account.credit(1000.0);
System.out.println(account.getBalance());
...
```

BankAccount method

```
public void credit(double amount) {
   setBalance(getBalance() + amount);
}
```

Overloading methods

- The same name may be used for many different methods, as long as they have different signatures
  - This is known as overloading
- The println() method of System.out.println() has 10 different parameter declarations:
  - boolean, char[], char, double, float, int,long, Object, String, and one with no parameters
  - You do not need to use different method names (such as "printStringñ or "printDoubleñ) for each data type you may want to print

Overriding methods

- A method with a signature and return type identical to a method in the subclass overrides the method of the superclass

```
public class BankAccount {
     private float balance;
     public int getBalance() {
          return balance;
     }
}

public class InvestmentAccount extends BankAccount {
     private float cashAmount
     private float investmentAmount;
     public int getBalance() {
          return cashAmount + investmentAmount;
     }
}
```

main method

- An application cannot run unless at least one class has a main() method
- The JVM loads a class and starts execution by calling the main(String[] args) method
  - public: the method can be called by any object
  - static: no object need be created first
  - void: nothing will be returned from this method

```
public static void main(String[] args) {
   BankAccount account = new BankAccount();
   account.setOwner(args[0]);
   account.credit(Integer.parseInt(args[1]));
   System.out.println(account.getBalance());
   System.out.println(account.getOwner());
  }
```

Encapsulation

- Private state can only be accessed from methods in the class
- Mark fields as private to protect the state
  - Other objects must access private state through public methods

```
package com.megabank.models;

public class BankAccount {
 private String owner;
 private double balance = 0.0;
}
```

```
public String getOwner() {
 return owner;
}
```

Static members

- Static fields and methods belong to the class
  - Changing a value in one object of that class changes the value for all of the objects
- Static methods and fields can be accessed without instantiating the class
- Static methods and fields are declared using the static keyword

```
public class MyDate {
    public static long getMillisSinceEpoch() {
        …
    }
}
…
long millis = MyDate.getMillisSinceEpoch();
```

Final members

- A final field is a field which cannot be modified
  - This is the Java version of a constant
- Typically, constants associated with a class are declared as static final fields for easy access
  - A common convention is to use only uppercase letters in their names

```
public class MyDate {
    public static final long SECONDS_PER_YEAR =
        31536000;
    …
}
…
long years = MyDate.getMillisSinceEpoch() /
```