



## *Building Classes*

Unit 05



## ***Unit Objectives***

---

- To learn about
  - Java classes and their syntax
  - Methods and fields
  - Java class construction in Eclipse

## 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; we usually need to create an object, or **instance** of the class

## ***Class Declaration***

---

- A class declaration specifies a type
  - The identifier
    - Specifies the name of the class
  - The optional `extends` clause
    - Indicates the superclass
  - The optional `implements` clause
    - Lists the names of all the interfaces that the class implements

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

    // Class Body
}
```

## ***Class Modifiers***

---

- The declaration may include class modifiers, which affect how the class can be used.
  - Examples:
    - `public`, `abstract`, `final`
- `public` classes
  - 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
  - Provide common information for subclasses
  - Cannot be instantiated
- A class is declared `final` if it permits no subclasses.

## ***Memory Management in Java***

---

- Java does not use pointers
  - Memory addresses cannot be accidentally or maliciously overwritten
- Java virtual machine handles all memory management
  - Problems inherent in user allocated and de-allocated memory are avoided
  - Programmers do not have to keep track of the memory they allocate from the heap and explicitly deallocate it

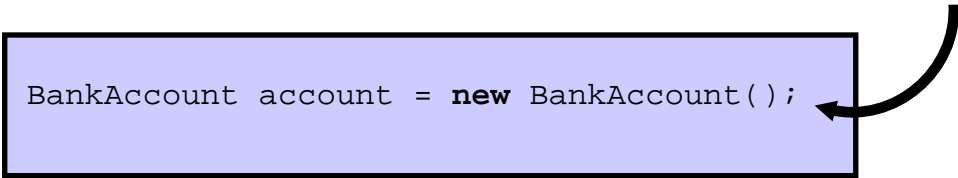
## Constructors

---

- A method that sets up a new instance of a class
  - The class body contains at least one constructor
- Use the `new` keyword with a **constructor** to create **instances** of a class

Class instantiation

```
BankAccount account = new BankAccount();
```



## More about Constructors

- Used to create and initialize objects
  - Always has the same name as the class it constructs (case-sensitive)
- 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



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

Constructor use



## ***Default Constructors***

---

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

## Overloading Constructors

- Overloading
  - When there are a number of constructors with different parameters
- Constructors are commonly overloaded to allow for different ways of initializing instances

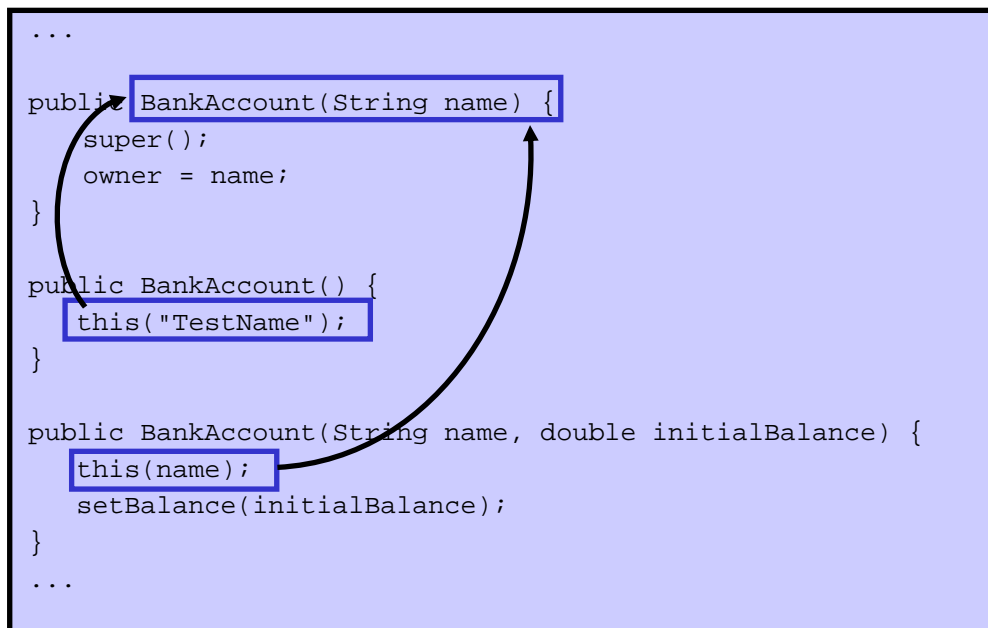
```
BankAccount new_account =  
    new BankAccount();  
  
BankAccount known_account =  
    new BankAccount(account_number);  
  
BankAccount named_account =  
    new BankAccount("My Checking Account");
```

## 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);  
}  
...  
...
```

A diagram illustrating constructor calls in Java. It shows three constructor methods for a class named BankAccount. The first constructor, public BankAccount(String name) {, is highlighted with a blue box. The second constructor, public BankAccount() {, contains the call this("TestName");, which is also highlighted with a blue box. The third constructor, public BankAccount(String name, double initialBalance) {, contains the call this(name);, which is also highlighted with a blue box. Two curved arrows originate from the highlighted this("TestName"); and this(name); lines and point back to the first constructor, indicating that these calls refer to the first constructor.

## ***Constructor Chaining***

---

- Constructor chaining
  - When one constructor invokes another within the class
- Chained constructor statements are in the form:
  - `this (argument list);`
  - The call is only allowed once per constructor
  - It must be the first line of code
- Do this to share code among constructors

## More on Constructor Chaining

- Superclass objects are built before the subclass
  - `super(argument list)` initializes superclass members
- The first line of your constructor can be:
  - `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 is done automatically by the JVM
  - The garbage collector reclaims 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
- Objects are eligible for garbage collection when:
  - They have no object references
  - Their references are out of scope
  - Objects have been assigned `null`
- The JVM decides when the garbage collector runs
  - Typically when memory is low
  - May not 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
  - Should only be used for special cases
    - Such as cleaning up memory allocation from native calls
  - Open sockets and files should be cleaned up during normal program flow before the object is dereferenced

## Fields

---

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

```
package com.megabank.models;

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

The diagram shows the following annotations:

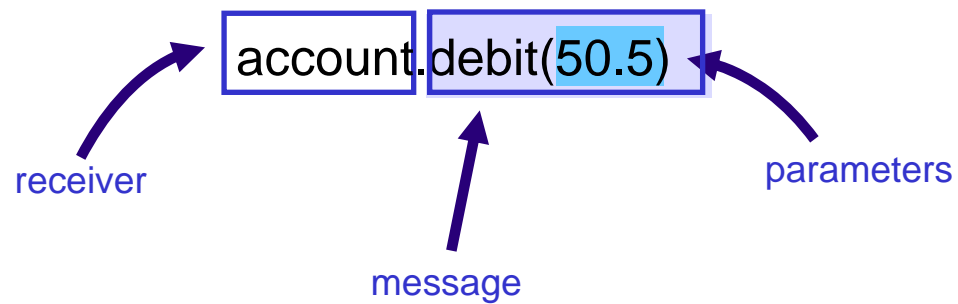
- access modifier**: points to the word `private` in the first field declaration.
- type**: points to the word `String` in the first field declaration.
- name**: points to the word `owner` in the first field declaration.

## Messages

---

- Use messages to invoke behavior in 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
  - The behavior of the class
    - All methods belong to a class

The diagram shows a Java method signature: `public void debit(double amount) {`. Four labels with arrows point to specific parts of the signature: 'access modifier' points to 'public', 'return type' points to 'void', 'method name' points to 'debit', and 'parameter list' points to '(double amount)'. The code continues with two lines of comments: `// Method body` and `// Java code that implements method behavior`, followed by a closing curly brace `}`.

```
public void debit(double amount) {  
    // Method body  
    // Java code that implements method behavior  
}
```

## 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

The diagram shows a code snippet: `public void credit(double amount) {`. The text `credit(double amount)` is enclosed in a brown rectangular box. An arrow labeled "method name" points to the word `credit`. An arrow labeled "argument type" points to the text `(double amount)`. An arrow labeled "signature" points to the entire boxed text `credit(double amount)`. The rest of the code snippet, `public void`, `...`, and `}`, is not highlighted.

```
public void credit(double amount) {  
    ...  
}
```

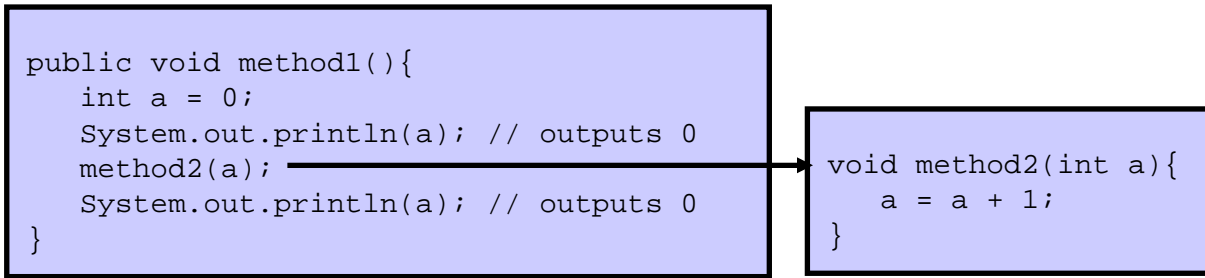
## Method Parameters

---

- Arguments (parameters) are passed by
  - Value for primitive types
  - 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
- There may be several return statements
  - Control goes back to the calling method upon executing a return

```
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 operator
  - The same operator is used for 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

---

- Signatures permit the same name to be used for many different methods
  - Known as *overloading*
- Two or more methods in the same class may have the same name but different parameters
- 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 don't need to use different method names, for example `"printString"`, `"printDouble"`, ...

## Overriding

---

- Override a method when a new implementation in a subclass is provided, instead of inheriting the method with the same signature from 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 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
- Constants associated with a class are typically 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() /  
    (1000*MyDate.SECONDS_PER_YEAR);
```

## ***Abstract Classes***

---

- **Abstract classes** cannot be instantiated – they are intended to be a superclass for other classes

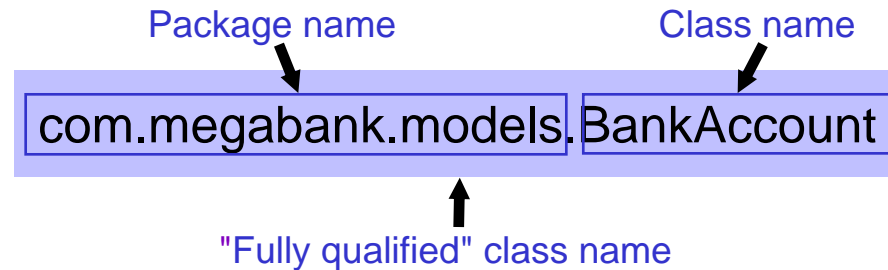
```
abstract class Learner {  
    public abstract String getName();  
    public abstract int getAge();  
    public int getMaxGrade() {  
        return getAge() - 5;  
    }  
}
```

- `abstract` methods have no implementation
- If a class has one or more `abstract` methods, it is `abstract`, and must be declared so
- **Concrete classes** have full implementations and can be instantiated

## Packages

---

- Classes can be grouped:
  - Logically, according to the model you are building
  - As sets designed to be used together
  - For convenience
- By convention, package names are in lower case
- Different packages can contain classes with the same name



## *Class Visibility*

---

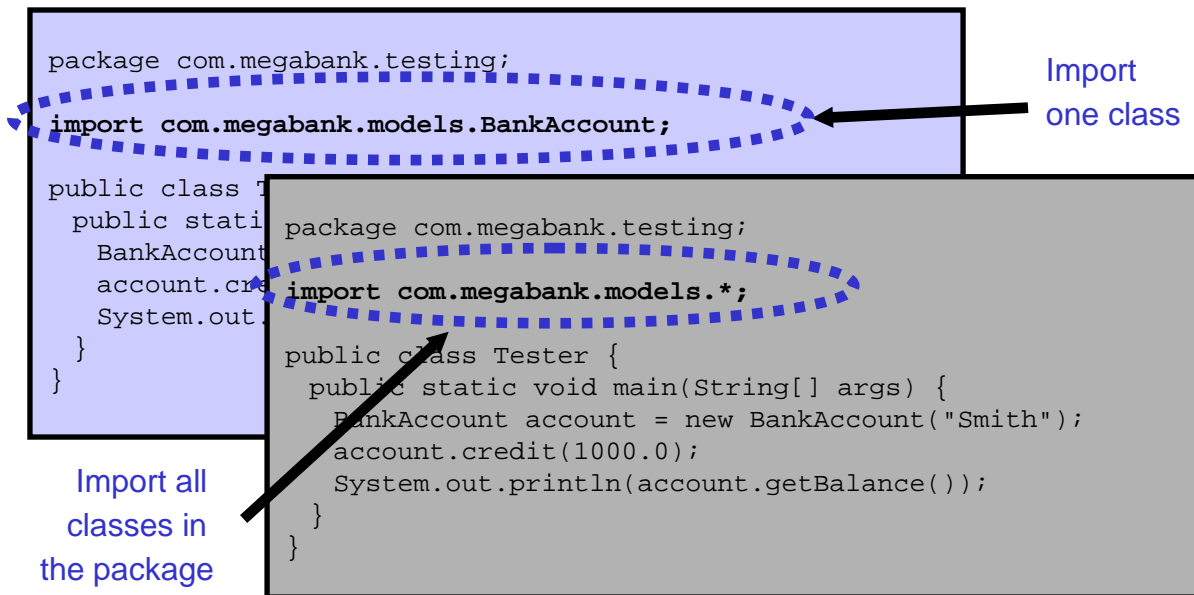
- Classes
  - Can reference other classes within the same package by class name only
  - Must provide the fully qualified name (including package) for classes defined in a different package
    - Below, Tester and BankAccount are defined in different packages

```
package com.megabank.testing;  
  
public class Tester {  
    public static void main(String[] args) {  
        com.megabank.models.BankAccount account1  
            = new com.megabank.models.BankAccount("Smith");  
        account1.credit(1000.0);  
        System.out.println(account1.getBalance());  
    }  
}
```

## Import Statement

---

- Include `import` statements to make other classes directly visible



## Java 1.4 Packages

---

- java.applet
- java.awt (\*)
- java.beans (\*)
- java.io
- java.lang (\*)
- java.math
- java.net
- java.nio (\*)
- java.rmi (\*)
- java.security (\*)
- java.sql
- java.text
- java.util (\*)
- javax.accessibility
- javax.crypto (\*)
- javax.imageio (\*)
- javax.naming (\*)
- javax.net (\*)
- javax.print (\*)
- javax.rmi (\*)
- javax.security (\*)
- javax.sound (\*)
- javax.sql
- javax.swing (\*)
- javax.transaction (\*)
- javax.xml (\*)
- org.ietf.jgss
- org.omg.CORBA (\*)
- org.omg.CosNaming (\*)
- org.omg.Dynamic (\*)
- org.omg.IOP (\*)
- org.omg.Messaging
- org.omg.PortableInterceptor (\*)
- org.omg.PortableServer (\*)
- org.omg.SendingContext
- org.omg.stub.java.rmi
- org.w3c.dom
- org.xml (\*)

## Core Java Packages

---

### •**java.lang**

- Provides classes that are fundamental to the design of the Java programming language
  - Includes wrapper classes, String and StringBuffer, Object, ...
  - Imported implicitly into all packages.

### •**java.util**

- Contains the collections framework, event model, date and time facilities, internationalization, and miscellaneous utility classes

### •**java.io**

- Provides for system input and output through data streams, serialization and the file system.

### •**java.math**

- Provides classes for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal)

### •**java.sql**

- Provides the API for accessing and processing data stored in a data source (usually a relational database)

### •**java.text**

- Provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages

## ***Sample Package: java.lang***

---

- Contains the following classes:
  - Basic Entities
    - Class Object Package System
  - Wrappers
    - Number Boolean Byte Character Double Float Integer Long Short Void
  - Character and String Manipulation
    - CharacterSubset CharacterUnicodeBlock String StringBuffer
  - Math Functions
    - Math StrictMath
  - Runtime Model
    - Process Runtime Thread ThreadGroup ThreadLocal  
InheritableThreadLocal RuntimePermission
  - JVM
    - ClassLoader Compiler SecurityManager
  - Exception Handling
    - StackTraceElement Throwable
- Also contains Interfaces, Exceptions and Errors

## ***Sample Class: String***

---

- **Sample Constructors:**

- String()
- String(byte[] bytes)
- String(byte[] bytes, int offset, int length)
- String(char[] value)
- String(char[] value, int offset, int length)
- String(String original)
- String(StringBuffer buffer)

- **Sample Methods:**

- char charAt(int index)
- boolean equals(Object anObject)
- int indexOf(String str)
- int length()
- boolean matches(String regex)
- String substring(int beginIndex, int endIndex)
- String toUpperCase()
- String trim()

## ***Sample Class: StringBuffer***

---

- Constructors:

- StringBuffer()
- StringBuffer(int length)
- StringBuffer(String str)

- Sample Methods:

- StringBuffer append(...)
- StringBuffer insert(...)
- StringBuffer delete(int start, int end)
- int length()
- StringBuffer reverse()
- String substring(int start, int end)
- String toString()

## ***Sample Class: Vector***

---

- Sample Constructors:**

- Vector()
- Vector(int initialCapacity)
- Vector(int initialCapacity, int Increment)

- Sample Methods:**

- boolean add(Object o)
- void clear()
- boolean contains(Object elem)
- Object elementAt(int index)
- Enumeration elements()
- boolean isEmpty()
- Object remove(int index)
- int size()
- Object[] toArray()

## ***What You Have Learned***

---

- How classes are built in Java
- Items covered include:
  - Class, method and field syntax
  - Static members
  - Packages and import statements
  - The representation of Java classes in Eclipse