Honors Computer Programming 1-2

Introduction To Chapter 2
Objects and Classes

Chapter Goals

- To understand the _concepts of classes and objects_
- To realize the _difference between objects and object references_
- To become familiar with the _process of implementing classes_
- To be able _to implement simple methods_
- To understand the _purpose and use of constructors_
- To understand how _to access instance fields and local variables_
- To appreciate the _importance of documentation comments_

Using and Constructing Objects

An object is an _entity_ in your program that you can _manipulate_ generally by calling _methods_ . For example, in Chapter 1 you saw how `System.out` was an object and you saw how to manipulate it using the `println` method. You should think of an object as a _black box_ with a public _interface_ (the methods that you call) and a hidden _implementation_ (the code and data to make the methods work).

Using and Constructing Objects

Different _objects_ support different _methods_ . For example, you can apply the `println` method to the `System.out` object but not to the string object _"Hello, World!"_ . It would be an error to call `"Hello, World!".println( );` . The reason is simple: `System.out` and `"Hello, World!"` belong to different _classes_ .

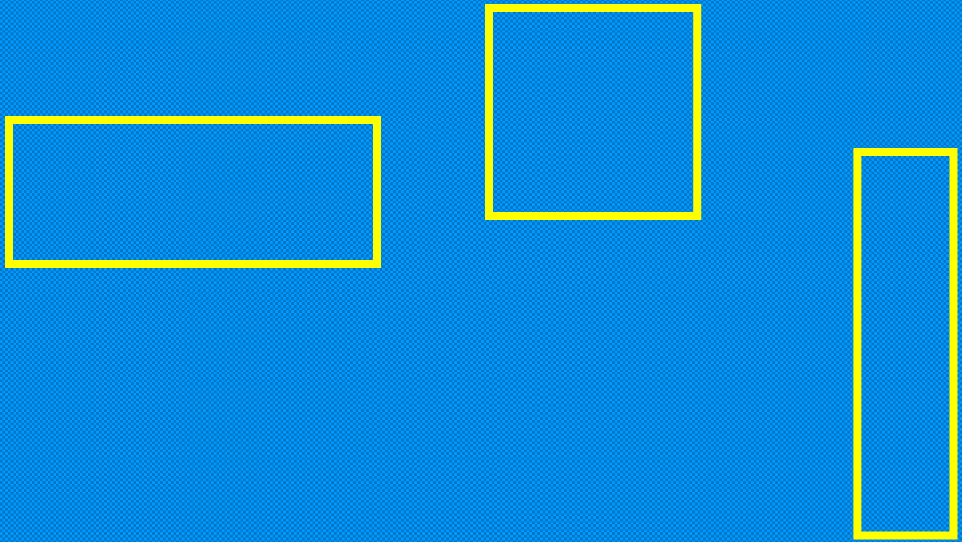
The `System.out` object is an object of the `PrintStream` class while `"Hello, World!"` is an object of the `String` class. You can apply the `println` method to any object of the `PrintStream` class but the `String` class does not support the `println` method.

Using and Constructing Objects

The String class does support a number of methods. For example, the `length` method counts the number of characters in a string. Thus `"Hello, World!".length( )` is okay and returns the number 13. You can test this by using the statement `System.out.println("Hello, World!".length( ));` in the `main` method.

Using and Constructing Objects

To see how to create new objects, let us turn to another class, the `Rectangle` class in the Java class library. Objects of type `Rectangle` describe ordinary rectangular shapes.



Using and Constructing Objects

Note that a `Rectangle` isn't a rectangular shape, it is a **set of numbers** that describe the rectangle. Each rectangle is described by the x- and y-coordinates of its **top left corner**, its **width**, and its **height**. To make a new rectangle, you need to specify these **four** values. For example, you can make a new rectangle with top left corner at (5, 10), width 20, and height 30 as follows:

`new Rectangle(5, 10, 20, 30)` .

new operator to make a new object

class name

parameters inside parenthesis

The `new` operator causes the creation of an object of type `Rectangle`. The process of creating a new object is called **construction**. The four values 5, 10, 20, and 30 are the **construction parameters**.

Rectangle

x = 5

y = 10

width = 20

height = 30

Using and Constructing Objects

Different classes will require _different__ construction parameters.

To construct a `Rectangle` object you supply 4 numbers that describe the _position_ and _size_ of the rectangle. To describe a `Car` object you might supply the _model name_ and _year_ .

Car
model name = _Ford Explorer_ year = _2004_

Using and Constructing Objects

Some classes let you __construct__ objects in different ways. You can also obtain a Rectangle object by supplying no parameters at all: `_new Rectangle( )_` . This constructs a (rather useless) rectangle with top left corner at `_(0, 0)_` , width `_0_` , and height `_0_` .

In general, to construct any object you do the following:

- use the `_new_` operator
- give the name of the _class_
- supply construction _parameters_ (if any) -- you are required to use _parenthesis_

Using and Constructing Objects

What can you do with a `Rectangle` object? Not much, _for now_. In chapter 4, you will learn how to _display_ rectangles and other shapes. At this time, you can pass a rectangle object to the _`System.out.println`_ method which will print a _description_ of the object onto the _console_ window.

So the command

```
System.out.println(new Rectangle(5, 10, 20, 30));
```

prints the line

```
java.awt.Rectangle[x=5, y=10, width=20, height=30]
```

Constructing Objects Summary

Syntax: **Object Construction**

```
new  ClassName ( parameters )
```

Example:

```
new  Rectangle(5, 10, 20, 30)
new  Car ("Ford Explorer",  2004)
```

Purpose:

To construct a new object, initialize it with the construction parameters, and return a reference to the constructed object.

Object Variables

To remember an object, you have to hold it in an _object variable_. A _variable_ is an item of information in _memory_ whose location is identified by a symbolic name. In Java, every variable has a particular _type_ that identifies the kind of information it can contain. You create a variable by giving its _type_ followed by a _name_ for the variable. For example, `Rectangle cerealBox;` defines a variable named `cerealBox`. The type of this variable is `Rectangle`.

Object Variables

Variable names must follow a few simple rules:

- Names can be made up of **letters**, **digits**, and the **underscore character**. They cannot start with a **number**.
- You cannot use other symbols such as **?** or **&** in variable names.
- **spaces** are not permitted inside names.
- You cannot use **reserved** words such as **public**. These words are reserved exclusively for their special **Java** meanings.
- Variable names are **case-sensitive**. That is, `cerealBox` and `Cerealbox` are **different** names.

Object Variables

In the declaration `Rectangle cerealBox;` the variable is not _initialized_ . That is, it doesn't have any _object location_ . To initialize a variable, you must use the _`new`_ operator which will create an _object_ and return its _location_ .

Object Variables

The following statement will declare and initialize the variable:

```
Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
```

a type — a variable — create a Rectangle object

The following diagram illustrates the difference between a declared variable and one that is initialized at declaration. Note that when the **new** operator is used, an **object** is created.

// declaration but no initialization

```
Rectangle cerealBox;
```

cerealBox =

no object is created

// declaration with initialization

```
Rectangle cerealBox =
  new Rectangle(5, 10, 20, 30);
```

cerealBox =

Rectangle

x = 5

y = 10

width = 20

height = 30

an object is created

Object Variables

An object location is called an **object reference**. When a variable contains the location of an object, we say that the variable **refers** to the object. It is important to remember that the `cerealBox` variable **does not contain** the object. It **refers** to the object.

You can also have two object v... same object. The additional declaration `Re`...`ealBox` makes the variable `r` refer to the sa...`alBox`.

> `cerealBox` and `r` refer to the same rectangle object

```
Rectangle cerealBox =
     new Rectangle(5, 10, 20, 30);
Rectangle r = cerealBox;
```

cerealBox =

r =

Rectangle

x = 5

y = 10

width = 20

height = 30

Object Variables

Usually your programs use objects in the following ways:

- construct an object with the _`new`_ operator
- store the reference to the _object_ in some _variable_
- call _methods_ on the object variable

Object Variables

The `Rectangle` class has over 50 methods. Consider the `translate` method which moves the rectangle a certain distance in the x- and y- directions. For example,

```
cerealBox.translate(15, 25);
```

moves the rectangle 15 units in the x-direction and 25 units in the y-direction. Moving a rectangle doesn't change its `width` or `height` but changes the top-left corner.

The code

```
Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
cerealBox.translate(15, 25);
System.out.println(cerealBox);
```

prints

```
java.awt.Rectangle[x=20, y=35, width=20, height=30]
```

Object Variables

Lets turn this into a complete program. You need to carry out three steps:

- invent a new class, say _`MethodTest`_
- supply a _`main`_ method
- place _instructions_ inside the main method

For this program, you need to carry out another step in addition to those: you need to _import_ the `Rectangle` class from a _package_. A package is a collection of _classes_ with a related purpose. The `Rectangle` class belongs to the package _`java.awt`_ where `awt` stands for _Abstract Windowing Toolkit_. To use this package, place the line _`import java.awt.Rectangle;`_ at the top of the program. Why didn't you have to import the _`System`_ and _`String`_ classes that were used in the Hello program? These classes are in the _`java.lang`_ package and all classes from this package are _automatically_ imported so you don't have to import them yourself.

Object Variables

Complete the comments:

```
import java.awt.Rectangle; // include the Rectangle package

public class MoveTest  // file must be named MoveTest.java
{
   public static void main(String[ ] args)
   {
      Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
      cerealBox.translate(15, 25); // move the rectangle
      System.out.println(cerealBox);  // print moved rectangle
   }
}
```

make a new Rectangle object

Object Variables

A common error is illustrated by the code at the right.

```
Rectangle  cerealBox;
cerealBox.translate(15, 25);
```

The first line creates a variable named `cerealBox` but does not use the `new` operator to create a `Rectangle` object. The second line attempts to move the rectangle but there is no rectangle to move. The compiler will complain that you are trying to use an uninitialized variable.

The remedy is to initialize the variable either using a new object:

```
Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
```

or to use an existing object:

```
Rectangle cerealBox = anotherRectangle;
```

Object Variables

Some programmers use a shortcut when importing packages. You can import _all_ classes from a package with a statement such as `import packageName.*;`. For example, in the program above you could use `import java.awt.*;`. We will not use this statement in this course and instead import the specific package with a statement such as `import java.awt.Rectangle;`.

Defining a Class

In this section we will learn how to define your own class. This first class will contain a single method.

```
public class Greeter
{
  public String sayHello( )
  {
    String message = "Hello,World!";
    return message;
  }
}
```

A method definition contains several parts:

- an access specifier (such as `public`)
- the return type of the method (such as `String`)
- the name of the method (such as `sayHello`)
- a list of parameters of the method enclosed in parenthesis (the sayHello method has no parameters)
- the body of the method (a sequence of statements enclosed in braces)

Defining a Class

```
public class Greeter
{
  public String sayHello( )
  {
    String message = "Hello,World!";
    return message;
  }
}
```

The access specifier controls which other methods can call this method. Most methods should be declared as public. That way all methods in your program can call them.

The return type is the type of value that the method returns to its caller. The `sayHello` method returns an object of type `String` namely `"Hello, World!"`. Some methods just execute some statements without returning a value. These methods are tagged with a return type of `void`.

Defining a Class

Many methods depend on _other_ information. For example, the _`translate`_ method of the _`Rectangle`_ class needs to know how far you want to move the rectangle horizontally and vertically. These items are called the _parameters_ of the method. Each parameter is a _variable_ with a _type_ and a _name_. Parameter variables are separated by _commas_.

```
public class Rectangle
{
   ...
   public void translate(int x, int y)
   {
     method body
   }
   ...
}
```

Defining a Class

The method body contains statements the method executes. The `sayHello` method body contains two statements.

```
public class Greeter
{
   public String sayHello( )
   {
      String message = "Hello,World!";
      return message;
   }
}
```

The first statement initializes a `String` variable with a `String` object:

```
String message = "Hello,World!";
```

The second statement is a special statement that terminates the method:

```
return message;
```

Defining a Class

```
public class Greeter
{
   public String sayHello( )
   {
      String message = "Hello,World!";
      return mess
   }
}
```

return type is different than void

When the **second** statement is executed, the method **exits** . If the method has a return type other than **void** , then the **return** statement must contain a **return value** , namely the value that the method sends back to its **caller** . The `sayHello` method returns the **object reference** stored in **message** -- that is, a reference to the **"Hello, World!"** string object.

Testing a Class

```
public class Greeter
{
  public String sayHello( )
  {
    String message = "Hello,World!";
    return message;
  }
}
```

The `Greeter` class can be __compiled__ but it cannot be __executed__ since it doesn't have a __`main`__ method. That is normal -- most classes don't have a main method. But you can write a __test class__. A test class typically carries out the following steps:

- construct one or more __objects__ of the class being tested
- invoke one or more __methods__
- print out one or more __results__

Testing a Class

The GreeterTest class tests the Greeter class. Notice that the main method of GreeterTest constructs an object of type Greeter using the new operator, invokes the sayHello method, and displays the result on the console.

```
public class GreeterTest
{
   public static void main(String[ ] args)
   {
      Greeter  worldGreeter = new Greeter( );
      System.out.println(worldGreeter.sayHello( ));
   }
}
```

```
public class Greeter
{
   public String sayHello( )
   {
      String message = "Hello,World!";
      return message;
   }
}
```

Testing a Class

```
public class GreeterTest
{
  public static void main(String[ ] args)
  {
    Greeter  worldGreeter = new Greeter( );
    System.out.println(worldGreeter.sayHello( ));
  }
}
```

file Greeter.java , you need to combine these two classes.

file GreeterTest.java

```
public class Greeter
{
  public String sayHello( )
  {
    String message = "Hello,World!";
    return message;
  }
}
```

- make two files, one for each class
- compile both files
- run the test program

Instance Fields

At this time all objects of type _`Greeter`_ would act the same way. Suppose you declare a `Greeter` object:

```
Greeter greeter1 = new Greeter( );
```

and then create a second `Greeter` object:

```
Greeter greeter2 = new Greeter( );
```

Then both `greeter1` and `greeter2` would return the _same_ result when you call the `sayHello` method. In order for each `Greeter` object to return a unique result, each object must _store state_.

The ***state of an object*** is the _set of values_ that determine how an object reacts to _method calls_.

Instance Fields

An object stores its state in one or more _variables_ called _instance fields_. The declaration at the right shows an ***instance field*** of the `Greeter` class called _`name`_. An instance field consists of the following parts:

- an access specifier (usually _`private`_)
- the type of the variable (such as _`String`_)
- the name of the variable (such as _`name`_)

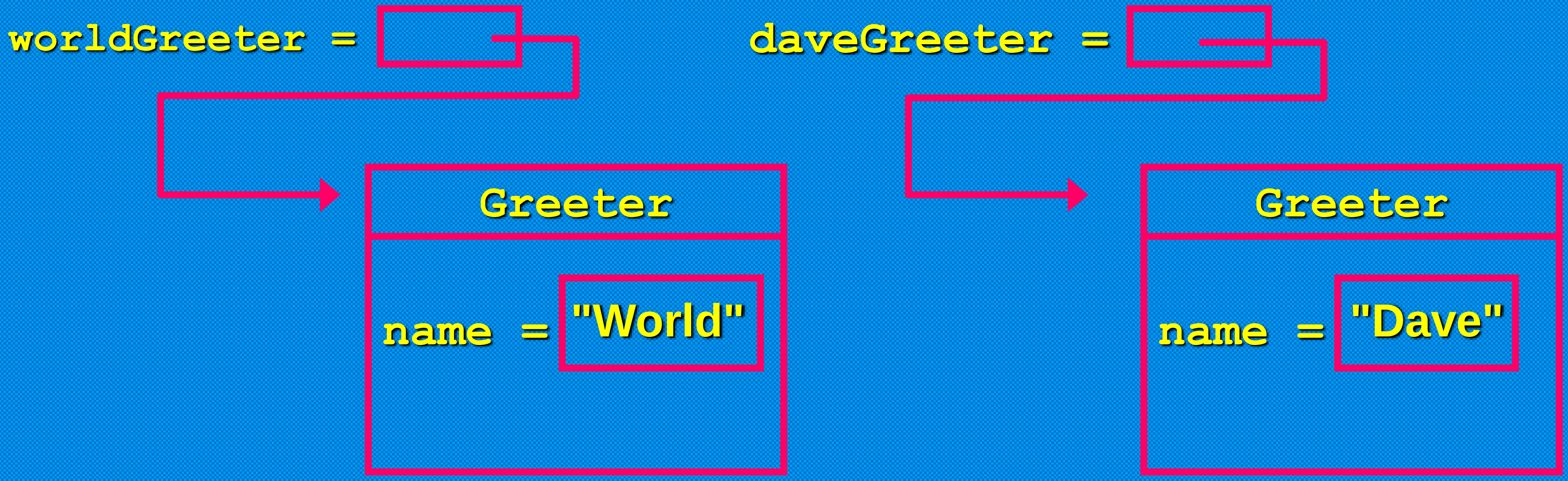
```
public class Greeter
{
  ...
  private String name;
}
```

Instance Fields

Each object of a class has its own set of __instance fields__.

For example, if `worldGreeter` and `daveGreeter` are two objects of the `Greeter` class, then each object has its own `name` field called `worldGreeter.name` and `daveGreeter.name`.

```
public class Greeter
{
   ...
   private String name;
}
```



Instance Fields

Instance fields are generally declared with the _access specifier_ as _private_ . That specifier means that they can only be accessed by methods of the _`Greeter` class_ , not by any other method. In particular, the _`name`_ variable can only be accessed by the _`sayHello`_ method.

In other words, if the _instance fields_ are declared as _private_ then all data access must occur though the _public_ methods. Thus the instance fields are effectively _hidden_ from the programmer who uses a class. They would only be of concern to the programmer who _implemented_ the class. The process of _hiding the data_ and providing methods for _data access_ is called _encapsulation_ . We will always make instance fields _private_ in this course.

Instance Fields

Since the name instance field is _private_ you cannot access the instance field in another class. Note the error in the revised `GreeterTest`.

```
public class GreeterTest
{
  public static void main(String[ ] args)
  {
    ...
    System.out.println(daveGreeter.name);   // error
  }
}
```

you can access the `name` instance field in `Greeter`
but you cannot access it in `GreeterTest`

Instance Fields

Only the sayHello method can access the private name variable. If we later add other methods to the Greeter class, such as a goodBye method, then those methods can access the private data as well. An improved sayHello method of the Greeter class is shown below.

```
public String sayHello( )
{
  String message = "Hello, " + name + "!";
  return message;
}
```

Instance Fields

```
public String sayHello( )
{
  String message = "Hello, " + name + "!";
  return message;
}
```

The __+__ symbol denotes __string concatenation__ an operation that forms a new string by pasting shorter strings together, one after another. This method __computes__ a string message by combining three strings: __`"Hello, "`__ plus the string contained in __`name`__ plus the string consisting of an __exclamation point__. If the name variable refers to the string `"Dave"`, then the resulting string is __`"Hello, Dave!"`__.

Instance Fields

```
public String sayHello( )
{
   String message = "Hello, " + name + "!";
   return message;
}
```

Note that this method uses two separate object variables: the local variable message and the instance field name. A local variable belongs to an individual method and you can only use it inside the method. An instance field belongs to a class and you can use it in all methods of the class.

`message` is declared locally within `sayHello`.
As a result, it can only be used within `sayHello`.

Constructors

To complete the improved `Greeter` class, we need to be able to __construct__ objects with different values for the __`name`__ instance field. We want to specify the name when constructing the object:

```
Greeter worldGreeter = new Greeter("World");
```

and

```
Greeter daveGreeter = new Greeter("Dave");
```

To accomplish this, we need to supply a __constructor__ in the class definition. A constructor specifies how an object should be __initialized__. The code for the constructor is shown below.

```
public Greeter(String aName)
{
   name = aName;
}
```

Constructors

```
public Greeter(String aName)
{
    name = aName;
}
```

no return type here

A constructor always has the same name as the <u>class of the objects it constructs</u>. Similar to methods, constructors are generally declared as <u>public</u>. Unlike methods though constructors do not have <u>return types</u>.

Constructors are not <u>methods</u>. You cannot invoke a constructor on an <u>existing object</u>. For example, the call:

```
worldGreeter.Greeter("World!");   // error
```

is illegal. You can only use the constructor in combination with the <u>`new`</u> operator.

Constructors

The code below is the enhanced `Greeter` class and the enhanced `GreeterTest` class whose purpose is to make sure the __Greeter__ class works correctly.

```
public class Greeter
{
  public Greeter(String aName)
  {
    name = aName;
  }
  public String sayHello( )
  {
    String message = "Hello, "
                     + name + "!";
    return message;
  }
  private String name;
}
```

```
public class GreeterTest
{
  public static void main(String[ ] args)
  {
    Greeter worldGreeter =
                  new Greeter("World");
    System.out.println
             (worldGreeter.sayHello( ));
    Greeter daveGreeter =
                  new Greeter("Dave");
    System.out.println
             (daveGreeter.sayHello( ));
  }
}
```

instance field for the Greeter class

Create a new Greeter object called daveGreeter

call the sayHello method using the daveGreeter object

main method for the GreeterTest class

pass the parameter "Dave" into the constructor

Designing the Public Interface of a Class

In this section we will create a class that describes the behavior of a bank account. Before you can start programming, you need to understand how the __objects__ of your class behave. Consider the kind of operations you can carry out with your bank account:

- __deposit__ money
- __withdraw__ money
- get the current __balance__

Designing the Public Interface of a Class

In Java, these operations are expressed as _method calls_ .

If the variable `harrysChecking` contains a _reference_ to a `BankAccount` then you will want to call methods such as the following:

- `harrysChecking.deposit(2000);` // deposit $2000
- `harrysChecking.withdraw(500);` // withdraw $500
- `System.out.println(harrysChecking.getBalance( ));` // print the balance

That is, the `BankAccount` class should define three methods: _`deposit`_ , _`withdraw`_ , and _`getBalance`_ .

Designing the Public Interface of a Class

Next, you need to determine the _parameters_ and _return types_ of these methods. As you can see from the samples, the `deposit` and `withdraw` methods receive a _number (dollar amount)_ and return _no value_. The `getBalance` method has _no parameter_ and returns _a number_.

Java has several number types that you will learn about in the next chapter. The most flexible _number type_ is called _`double`_. Examples of doubles are `250`, `6.75`, or `-0.3333333`.

Designing the Public Interface of a Class

Now that you know you can use `double` for the number type, you can write down the methods of the `BankAccount` class:

- `public void deposit(double amount)`
- `public void withdraw(double amount)`
- `public double getBalance( )`

void means return no value

a parameter of type double

return a number

no parameters

Designing the Public Interface of a Class

How do we want to __construct__ a bank account? It seems reasonable that the statement

```
BankAccount harrysChecking = new BankAccount( );
```

should construct an account with a __zero__ balance. What if we want to start out with another balance? A second __constructor__ would be helpful that sets the balance to an initial value:

```
BankAccount harrysChecking = new BankAccount(5000);
```

Designing the Public Interface of a Class

That gives us two constructors: `public BankAccount( )` and `public BankAccount(double initialBalance)` The compiler figures out which constructor to call by looking at the _parameters_. For example, if you call `new BankAccount( )` then the compiler picks the first constructor. If you call `new BankAccount(5000)` then the compiler picks the second constructor. But if you call `new BankAccount("lotsa moolah")` then the compiler generates an error message since there is no constructor with a _string_ parameter.

Designing the Public Interface of a Class

These constructors and methods form the **public interface** of the class. Here is how you can:

// transfer money from one account to another

```
double transferAmount = 500;
momsSavings.withdraw(transferAmount);
harrysChecking.deposit(transferAmount);
```

// add interest to a savings account

```
double interestRate = 5;     // 5% interest
double interestAmount = momsSavings.getBalance( )
                             * interestRate / 100;
momsSavings.deposit(interestAmount);
```

Designing the Public Interface of a Class

As you can see, you can use _public objects_ of the `BankAccount` class to carry out important tasks, without knowing how the `BankAccount` objects _store their data_ or how the `BankAccount` methods _do their work_. This process of determining the feature set of a class is called _abstraction_. When you design the _public interface_ for a class, you need to find what operations are essential to manipulate objects in your program.

Overloading

When the same name is used for more than one method or constructor, the name is _overloaded_ . This is common for _constructors_ since all constructors have the same name -- the _name of the class_ . In Java, you can overload _methods_ and _constructors_ provided the _parameter types_ are different.

Overloading

The code at the right shows that the class `PrintStream` defines many methods, all called `println`, to print various number types and to print objects.

```
class PrintStream
{
  public void println(String s) {...}
  public void println(double a) {...}
  ...
}
```

When the `println` method is called with a statement such as

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

the compiler looks at the type of `x`.

If `x` is a `String`, the first method is called. If `x` is a `double`, the second method is called. If `x` does not match the parameter types for any of the methods, a compiler error is generated.

For overloading purposes, the type of the return value does not matter. You cannot have two methods with identical names and parameter types but different return values.

Specifying the Implementation of a Class

Its now time to supply the __implementation__. You already know you need to supply a class with these ingredients:

```
public class BankAccount
{
    constructors
    methods
    instance fields
}
```

We already know the methods and constructors we want. The instance fields are used to store the __object state__. In this case the state is the account __balance__. So a single instance field is sufficient: `private double balance;`.

Specifying the Implementation of a Class

Note that the instance field is declared with the access specifier `private` . That means that the balance can be accessed only by constructors and methods of the same class and not by any method or constructor of a different class. How the account balance is maintained is a private implementation detail of the class. Recall that the practice of hiding the implementation details and providing methods for data access is called encapsulation .

The primary benefit of the encapsulation mechanism is the guarantee that an object cannot accidentally be put into an incorrect state. For example, suppose you want to make sure that a bank account is never overdrawn . You can implement the `withdraw` method so that it refuses to carry out a withdrawal that would result in a negative balance .

Specifying the Implementation of a Class

Here is the **deposit** method

```
public void deposit(double amount)
{
    double newBalance = balance + amount;
    balance = newBalance;
}
```

local variable

set instance field to a new value

instance field

parameter

and here is the default constructor (the one with no parameters)

```
public BankAccount( )
{
    balance = 0;
}
```

set new account balance to zero

no parameters

The code for the **BankAccount** class is on the handout.

Specifying the Implementation of a Class

A common error is to try to __reset__ an object by calling a constructor. The constructor is invoked only when an object is __first created__.

Note the code below which contains an error.

```
BankAccount harrysChecking = new BankAccount( );
harrysChecking.withdraw(500);
harrysChecking.BankAccount( );     // error
```

The constructor sets a __new__ account to a __zero__ balance, but you cannot invoke the constructor on an __existing__ object. The remedy is simple: Make a __new__ object and overwrite the current one:

```
harrysChecking = new BankAccount( );  // ok
```

Variable Types

We have considered three types of variables in this chapter: _instance fields_ , _local variables_ , and _parameter variables_ . These variables are similar to each other but have some _differences_.

The first difference is _lifetime_ . An instance field belongs to an _object_ . Each object has its own copy of every instance field. When an object is constructed, its instance fields are _created_ . They stay alive until no _method_ uses the object any more.

Local and parameter variables belong to a _method_ . When the method starts, these variables _come to life_ . When the method exits, these variables _die_ .

Variable Types

The second difference between local and instance variables is _initialization_. You must initialize all _local_ variables. If you don't, the _compiler_ complains when you try to use it. Parameter variables are initialized with the values that are _supplied in the method call_.

Instance fields are initialized with a _default value_ if you don't explicitly set them in a constructor. Instance fields that are numbers are initialized to _0_. Object references are initialized to a special value called _`null`_. If an object reference is null, then it refers to _no object_. It is a matter of good style to initialize _every_ instance field in _every_ constructor.

Variable Types

Consider the "lazy" constructor shown at the right for the `Greeter` class. Since `name` is an *instance field* of type `String` which is an object, when the constructor does not initialize it, `name` will have a default value of `null`.

```
public class Greeter
{
  public Greeter( ) { }   // do nothing
  ...
  private  String  name;
}
```

When you call the `sayHello` method it will return `"Hello, null!"`.

Explicit and Implicit Method Parameters

Have a look at a particular invocation of the `deposit` method:

```
momsSavings.deposit(500);
```

Also look at the code for the `deposit` method at the right.

```
public void deposit(double amount)
{
   double newBalance = balance + amount;
   balance = newBalance;
}
```

The parameter variable `amount` is set to 500 when the `deposit` method starts. Since we deposit the money into `momsSavings`, `balance` must mean `momsSavings.balance`

So the call to `deposit` depends on two variables: the object to which `momsSavings` refers and the value 500.

Explicit and Implicit Method Parameters

```
public void deposit(double amount)
{
  double newBalance = balance + amount;
  balance = newBalance;
}
```

The _amount_ parameter inside the parenthesis is called the _explicit_ parameter because it is explicitly named in the method definition. However, the reference to the `BankAccount` object is not explicit in the method definition -- it is called the _implicit_ parameter.

If you need to, you can access the implicit parameter with the keyword _this_. For example, in the preceding method `this` was set to _momsSavings_ and amount to _500_. The statement

```
double newBalance = balance + amount;
```

actually means

```
double newBalance = this.balance + amount;
```