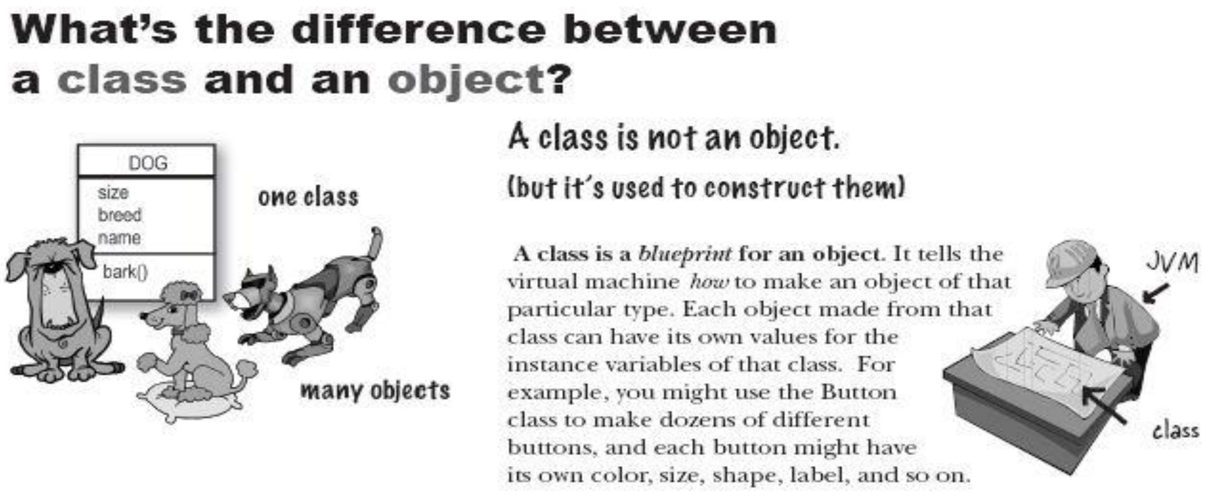
Description

This week, we'll do more with classes, learning about reference variables, and how to declare and work with classes and objects. Beside using a class that already exist in the Java API, we'll also create our own classes. We'll learn about constructor and its purpose, adding service methods and more about **variables** - their scope and lifetime and how they are passed to methods.

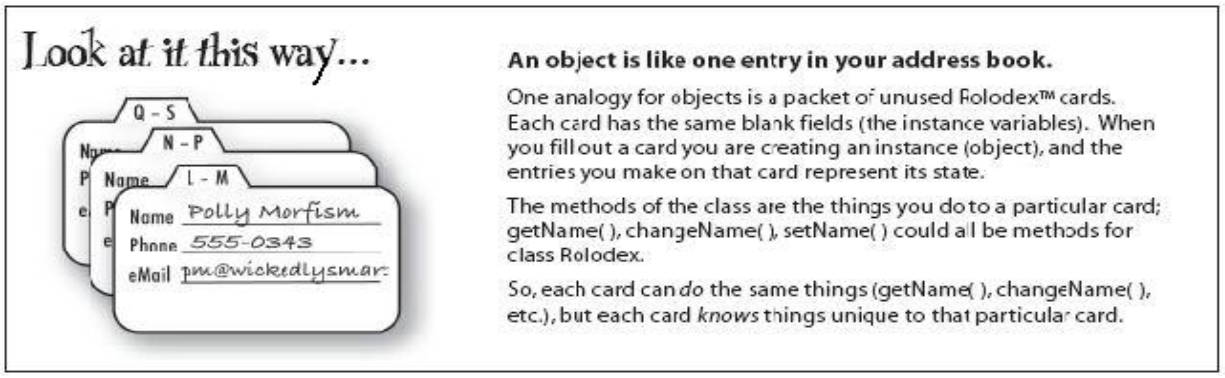
Classes and Objects



What's the difference between a class and an object?

A class is not an object.
(but it's used to construct them)

A class is a ***blueprint*** **for an object.** It tells the virtual machine *how* to make an object of that particular type. Each object made from that class can have its own values for the instance variables of that class. For example, you might use the Button class to make dozens of different buttons, and each button might have its own color, size, shape, label, and so on.



Look at it this way...

An object is like one entry in your address book.

One analogy for objects is a packet of unused Rolodex™ cards. Each card has the same blank fields (the instance variables). When you fill out a card you are creating an instance (object), and the entries you make on that card represent its state.

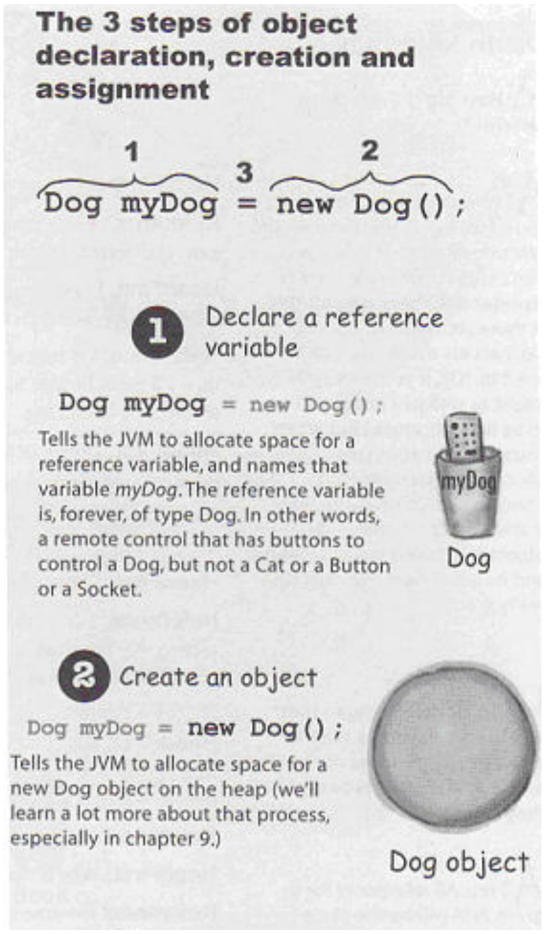
The methods of the class are the things you do to a particular card; getName( ), changeName( ), setName( ) could all be methods for class Rolodex.

So, each card can *do* the same things (getName( ), changeName( ), etc.), but each card *knows* things unique to that particular card.

The most common use of a class is to model a "real-world" concept or thing. We might write a class to represent a game player, an alien, a student, a course, a bank account, a product that our company sells, an order from a customer for that product, etc. Classes have fields and methods.

Remember that a **class** is not an **object**, *a* ***class*** *is a blueprint or template for an object*. Other common analogies are: the class is the recipe and the cookies (objects) are baked using the recipe, a class is like an address book and objects are like entries in the book. While we're at it, think of **instance** as another way of saying object. *An* ***object*** *is an instance of a class.*

Objects



The 3 steps of object declaration, creation and assignment

$\underbrace{\texttt{Dog myDog}}_{1} \overset{3}{=} \underbrace{\texttt{new Dog();}}_{2}$

1 Declare a reference variable

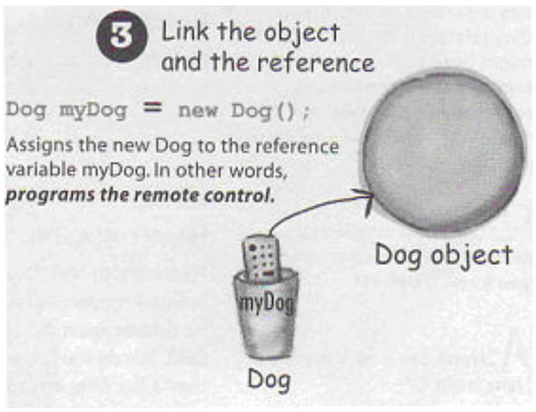
```
Dog myDog = new Dog();
```

Tells the JVM to allocate space for a reference variable, and names that variable *myDog*. The reference variable is, forever, of type Dog. In other words, a remote control that has buttons to control a Dog, but not a Cat or a Button or a Socket.

2 Create an object

```
Dog myDog = new Dog();
```

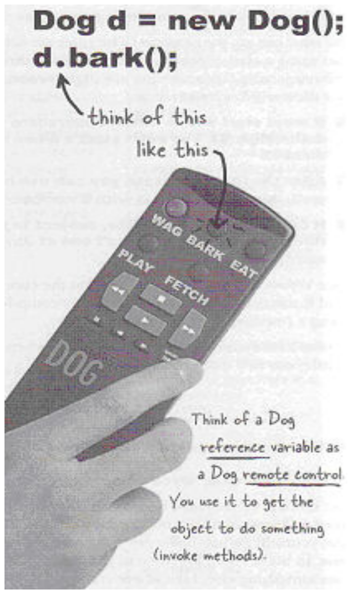
Tells the JVM to allocate space for a new Dog object on the heap (we'll learn a lot more about that process, especially in chapter 9.)



3 Link the object and the reference

```
Dog myDog = new Dog();
```

Assigns the new Dog to the reference variable myDog. In other words, ***programs the remote control.***



```
Dog d = new Dog();
d.bark();
```

Think of a Dog reference variable as a Dog remote control. You use it to get the object to do something (invoke methods).

So far, we've used two classes from the Java API: System.out and Math. When we work with them, we don't need to declare an object of the class (aka reference variable). This is because these two classes are different from other classes in the API, in that they have static methods. static is a keyword in Java that lets us use a class without declaring or using an object.

However, there are also classes in the Java API with no static methods. For these regular classes and for the classes we create ourselves, we must declare an object to use the class. **(Objects are also called reference variables, object reference variables, or class variables!).** Declaring a reference variable is also called **instantiating** an object or creating an **instance** of a class.

Like primitive variables, object reference variables are initialized with an explicit **new** operation. The keyword new allocates space for the object in memory. The brackets after the class name on the right side of the statement are also part of the object initialization. They cause a special method to be executed: the class constructor. The constructor in a class is usually used to set data values (fields) for the object.

The statement:

Rectangle myRectangle1;

is declaring a variable, in the same way that

int serialNumber;

declares a variable.

Remember that Java is a case-sensitive language. In the statements above, **"Rectangle"** is a class name, while "myRectangle1" is a variable name.

The expression

new Rectangle();

creates an *object* of the **Rectangle** class.

To use an object in your program, you must assign it to a variable of the appropriate type. The complete statement

Rectangle myRectangle1 = new Rectangle();

creates and initializes a **Rectangle** object and assigns it to the variable myRectangle1. In effect, this gives the object the name "myRectangle1".

If you declare a reference variable, but neglect to create and initialize the object by coding the **new**, it will also be null. If this happens, you will get NullPointerException.

<u>Scanner</u>

The Scanner class is used to get input from the user via the keyboard. Scanner is an example of a regular class (its methods are NOT static like System.out and Math), so have to declare a reference variable (object) to use it. We do this as follows:

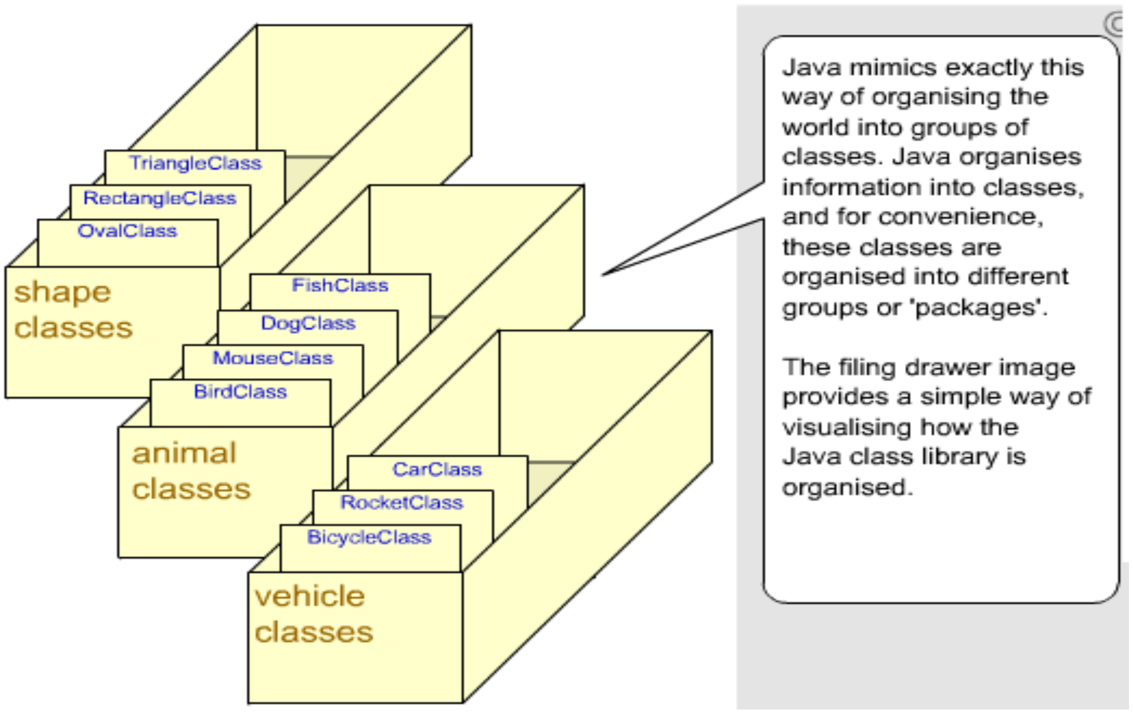
Scanner scanner = new Scanner(System.in);

In the case of Scanner, we pass a parameter to the Scanner class constructor that tells it to use the System.in device (the keyboard) as the input device where the user will enter data. The data the user keys will show in the Output window (the console for NetBeans) so we can see it.

<u>Import</u>

Packages and importing libraries

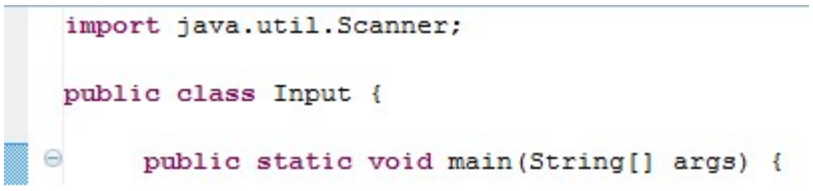
Here's the picture that explains the relationship between classes and packages in Java:


TriangleClass
RectangleClass
OvalClass
shape classes
FishClass
DogClass
MouseClass
BirdClass
animal classes
CarClass
RocketClass
BicycleClass
vehicle classes
Java mimics exactly this way of organising the world into groups of classes. Java organises information into classes, and for convenience, these classes are organised into different groups or 'packages'.
The filing drawer image provides a simple way of visualising how the Java class library is organised.

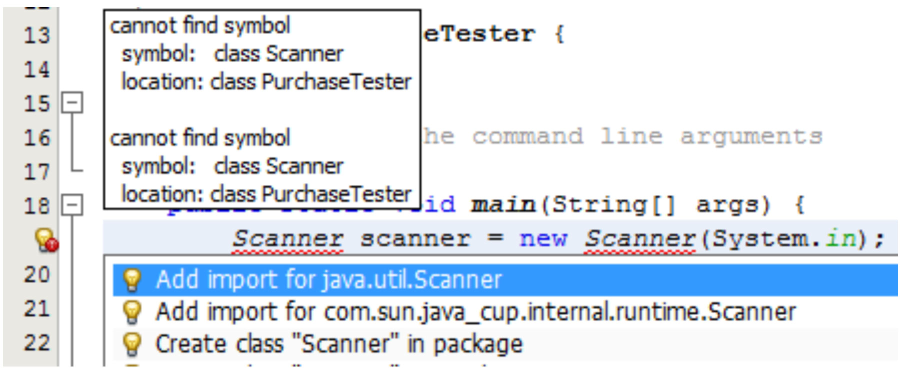
The Java API (Application Programming Interface) provides you with a library of hundreds of classes. We have already seen two of these, System.out and Math, and Scanner is a third one. The library classes are organized in groups called packages. Some of the most used library classes, including System.out and Math, are in a package called java.lang. Classes in this package are always available to your projects. Library classes from other packages, however, must be *imported* before you can use them. To get input from the user via the Output window in NetBeans or the console in text editor applications, we need the Scanner class. The Scanner class is in a package called java.util, so you must include the import statement that follows in any program that needs to use this class:

import java.util.Scanner;

Note that this statement is placed **BEFORE** the class header in your class.


```
import java.util.Scanner;

public class Input {

    public static void main(String[] args) {
```

Forgetting to import a library class is a common error, so it's good to be able to recognize this message when you see it. If you click on the Import 'Scanner' message in the pop-up box, NetBeans will add the import for you:



```
cannot find symbol
  symbol:   class Scanner
  location: class PurchaseTester

cannot find symbol
  symbol:   class Scanner
  location: class PurchaseTester
```

```
eTester {
he command line arguments
id main(String[] args) {
    Scanner scanner = new Scanner(System.in);
```

```
Add import for java.util.Scanner
Add import for com.sun.java_cup.internal.runtime.Scanner
Create class "Scanner" in package
```

If you are using several classes from the same package, you can import the entire package by writing

import java.util.*;

The asterisk is a *wild card* that represents all the classes in the package java.util.

Scanner Methods

Consider the following statements:

```
System.out.println("Please enter an integer: ");
int int1 = scanner.nextInt();
```

The first statement just tells the user what we need them to do. In the second statement, the right-hand side:

```
scanner.nextInt();
```

is *calling a method*, named nextInt, that acts on the object we had previously assigned to the reference variable named scanner. Unlike primitives, reference variables (objects) refer to classes that contain methods. To work with reference variables, you send messages to objects, or call methods.

The nextInt method waits for the user to enter something in the Output window and press enter. The value entered by the user is *returned* by the method. Returned means that the value is sent back by the method, and we need to do something with the value. In the statement above, we assign it to an integer called int1.

Let's say the user enters 7. In this case, the statement resolves to int1 = 7;

```
scanner.next();
```

Scanner's next method will scan only up to a space, if you want to scan a whole line including spaces, you need nextLine.

```
scanner.nextLine();
```

If you use nextLine after another scanner method such as nextDouble, there will be a linefeed left in the buffer because nextDouble doesn't read the linefeed, only the data before it. In this case, you need an extra nextLine call to read the linefeed that's left in the buffer first before you use the nextLine to read the new line of data.

Our First Class

Let's create a class of our own, a BankAccount class (remember that class names always start with a capital letter). It is up to us, the designers of the class, to decide what parts of the real-world thing we want to model in our class, and what we want to leave out. A class that represents a BankAccount could include information on the balance of the BankAccount, its interest rate, whether it's a savings or chequing account, whether it has a minimum balance, etc. To keep things simple, we will only model two fields (aka property, attribute, instance variable) of the BankAccount: its balance and accountId.

BankAccount
Attributes private int accountID private double balance
Operations public BankAccount() public BankAccount(int id,double newBalance) public double getBalance() public int getAccountID() public void setBalance(double balance) public void setAccountID(int newID) public void deposit(double amount) public void withdraw(double amount)

We could also include many methods in our class. Again, it's up to us to decide which ones should be included. The UML (Unified Modelling Language) diagram shows what our BankAccount class will contain.

When we work with classes, we will create our classes as separate files in our NetBeans project. This way, they can be reused again in other projects later. The source code for two classes is shown below. The first defines our BankAccount class, and the other, called BankAccountTest, contains the main method and makes use of the BankAccount class in an executable program. BankAccountTest's only purpose is to hold a main method.

```
/* A class to represent a bank account.
*/
public class BankAccount {
   // field (aka instance field , instance variable, attribute or property)
   private double balance;
   private int accountID;

   // getter (accessor)
   public double getAccountID() {
              return accountID;
   }
   // setter (mutator)
```

```
    public void setAccountID(double newID) {
                accountID = newID;
    }
    public double getBalance() {
                return balance;
    }
    public void setBalance(double newBalance) {
                balance = newBalance;
    }
}
```

```
/* A class to test our BankAccount class - contains main method
*/
public class BankAccountTester {

    public static void main(String[] args) {
           // Declare reference variables to refer to BankAccount objects
           BankAccount account1;
           BankAccount account2;

           account1 = new BankAccount();
           account2 = new BankAccount();

           // Call methods to change the balance in account1 and account2
           account1.setBalance(10000);
           account2.setBalance(500);

           System.out.println("account1 balance " + account1.getBalance());
           System.out.println("account2 balance " + account2.getBalance());
      }
}
```

Here's what the output looks like:

```
account1 balance 10000.0
account2 balance 500.0
```

<u>Fields/Attributes</u>

There are two things that typically go inside the definition of a class like BankAccount: fields (properties) and methods. Also, let's not forget about constructors, which are a special type of method. Fields are also called properties, data, attributes, instance variables, or instance fields. Fields are the things an object KNOWS. ***Fields*** *are attributes or characteristics that describe an object*. If color is a field - an object knows its color.

In general, a class can have many fields, but because our BankAccount class is very simple, it only has two:

```
private double balance;
private int accountID;
```

A field is a variable. Like a variable, a field declaration always includes a data type (in this case, double) and a name (in this case, balance). Names for fields follow the same conventions as for other variables: they should begin with a lower case letter, and if they are more than one word long, then every word except the first one begins with an upper case letter.

Unlike the declaration of a variable, a field declaration also includes a modifier called an *access specifier*. There are four possible access specifiers: public, private, protected and nothing at all. **Except in very special circumstances, fields should have private access.** Private access means the field can be accessed only within the class it's declared in. This stops other objects from reaching in and modifying the value in the field. Get in the habit of making your fields private.

Note that leaving out the access specifier is not a syntax error. Your program will still compile and run correctly. This, however, gives what is called package access to your fields and it is **NOT** what you want. The following chart shows the access specifiers and their meanings in classes, methods and fields.

Modifier	**Classes and interfaces**	**Methods and variables**
default (no modifier)	Visible in its package	Visible to any class in the same package as its class
public	Visible anywhere	Visible anywhere
protected	N/A	Visible to any class in the same package as its class
private	Visible to the enclosing class only	Not visible to any other class

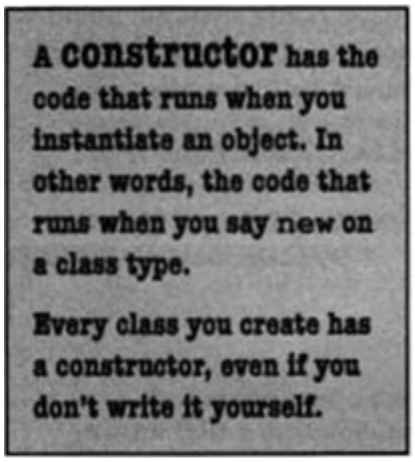
Fields are automatically given default values. Unless you explicitly assign other values to them, numeric primitive types will be initialized to zero, char to the Unicode value zero ('\u0000'), boolean to false, and class types (objects) to null. You can also assign a different default value at the time you declare a field.

Constructors

The purpose of a constructor is to create an *object* of a class.

When we create one specific BankAccount with a given balance, we call this a BankAccount *object*. We can also call this an *instance* of the class BankAccount, and the process of creating an object is called *instantiation*.

Let's have a look at the main method in our BankAccountTest class. In this method, we first declare two local variables whose data type is BankAccount.



A constructor has the code that runs when you instantiate an object. In other words, the code that runs when you say new on a class type.

Every class you create has a constructor, even if you don't write it yourself.

```
BankAccount account1;
BankAccount account2;
```

These variables can be used to refer to BankAccount objects, but we have not yet created any such objects.

We create a BankAccount object by calling the constructor of the class. The name of the constructor is always the same as the name of the class itself. Constructors are like methods in many ways. The constructor's name is always followed by parentheses and is preceded by the Java keyword "new".

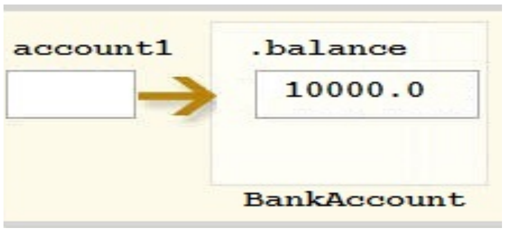
```
account1 = new BankAccount();
```

The constructor creates a new object of the class BankAccount. This means that space is allocated in memory for the object. Inside the object are all the fields specified in the class definition. In this case, there are only two fields, balance and accountId.

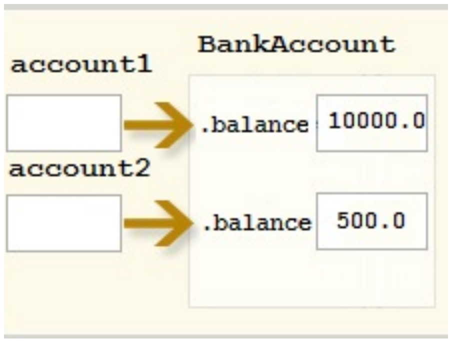
The most common task in a constructor is to initialize the fields of the object.

To be able to work with the newly created object later in our program, we need to assign it to a variable so that we can refer to it. So, a constructor call is usually found on the right-hand side of an assignment statement:

```
account1 = new BankAccount();
account1.setBalance(10000);
```



The variable account1 and the object are related, but they are two different things at different places in memory.



If we call the constructor of BankAccount a second time, this creates a second BankAccount object, at a different location in memory from the first object. Each object has its own set of fields. In this case, each BankAccount object has its own field called balance. The balance of account1 is separate from the balance of account2.

<u>Getters and Setters</u>

When we work with classes, we create our own methods to code the things an object can DO. ***Methods*** *are predefined actions that can be performed by an object.* Let's look at getter and setter methods.

The same way the state of an object is determined by the values of its fields, the **behaviour** of an object is determined by the methods it has available.

Getters and Setters

A class's methods often operate on its fields (properties, attributes or data). It's common for a class to have methods that read or write the values of the fields. Two of the methods in our BankAccount class are getBalance, which is an accessor (getter) method, and setBalance, which is a modifier (setter or mutator) method. These methods get and set the balance field - most classes have these two types of methods for each field.

```
 // getter or accessor method - retrieves the value of a field
  public double getBalance() {
     return balance;
  }

 // setter or mutator method - changes value of a field. Often edits to ensure value is valid

 public void setBalance(double newBalance) {
       if(balance >= 0){
          balance = newBalance;
       }
  }
```

These methods are always public so they can be called by other classes.

Getters

Getters retrieve the value of the field. They always have a return type the same as the datatype of the field they are returning. They do not have parameters, and usually just contain a return statement.

Setters

The setter changes the value of a field. It often includes code to make sure the value it's being changed to is valid. Setters are always void and they always take one argument, which is the datatype of the field they are modifying. The code always takes the parameter and assigns it to the field, thus changing its value.

Calling a method

We make use of our objects by calling methods that act on them. A method acts on one instance of the class in which the method is defined. In Java, when we call the method, we must precede the method name with the name of the object we want it to act upon, and a . (dot). We then write the method name, followed by parentheses, with arguments in the parentheses. The arguments in the method call must match (in number, data type, and order) the parameters in the method definition.

Four Things to remember about constructors

1. A constructor is the code that runs when somebody says new on a class type
   Duck d = new Duck();
2. A constructor must have the same name as the class, and no return type
   public Duck (int size) { }
3. If you put a constructor in your class, the compiler puts in a default constructor. The default constructor is always a no-arg constructor.
   public Duck() { }
4. You can have more than one constructor in your class, if the argument lists are different. Having more than one constructor in a class means you have overloaded constructors.
   public Duck() { }
   public Duck (int size) { }
   public Duck (String name) { }
   public Duck (String name, int size) { }

Overloading Constructors

An **overloaded method** *is a method with more than one version - each version has the same name but different arguments (parameters).* When you call an overloaded method, the compiler knows which version of the method to use based on the number and the data types of the parameters you pass to it. Here's the constructor we've already coded:

```
public BankAccount(int id, double newBalance) {
   balance = newBalance;
   accountId = id;    }
```

Rather than coding it this way, we are better off coding it to call our setter method. This way, any editing that has been included in our setter will be executed before the value of balance is modified:

```
public BankAccount(int id, double newBalance) {
     setAccountId(id);
     setBalance(newBalance);    }
```

On to overloading --> Constructors can be overloaded in the same way as methods: a class can have more than one constructor, as long as they all have different parameter lists. For example, we could add a second constructor to our BankAccount class:

```
public BankAccount() {
       setBalance(0.0);
```

```
        setAccountId(0);    }
```

We now have two constructors for our class, the original one with two parameters and another with no parameters.

If we do not define any constructor at all, then the compiler automatically creates one for us. This is called the *default constructor*. This constructor takes no arguments. It creates an object of the specified class, with all its instance fields initialized to their default values.

For example, if we had not defined any constructor for our BankAccount class, then the compiler would create a default constructor. We could call it by writing

```
account1 = new BankAccount();
```

This would create a BankAccount object with balance and accountId zero.

Defining Methods

To code methods in our classes, we must *create* or *declare* them. A method definition has a *header* line and a *body*.

```
public void deposit() {
```

Modifier

The modifier "public" is the *access specifier*. The great majority of methods have public access. Notice that class BankAccount also has public access, so other classes (such as the class BankAccountTester) can create objects of it. BankAccountTester is also public.

Return type

The next word is "void". This is called the *return type* of the method. **All methods must have a return type in the header line of their definition.** We've already used both void and non-void methods of java classes in our coding. Note that this is why constructors are special types of methods - they don't have a return type at all, not even void!

After a method has finished doing its work, it can send a value back to the program that called it. This is called the *return value*. Some methods do not return a value. In this case, their return type is void. If the method **does** return a value, then the data type of the return value must be indicated in the header of the method definition. In the method above, **if** the word "double" was there instead of "void" before "deposit" it would indicate that this method always returns a double value.

Method Names

The conventions for method names are the same as for variable names. **Method names should always start with a lower case letter.** If the name has more than one word, these are concatenated together. Each word except the first should start with an upper case letter.

These naming conventions are not rules of the language. Naming conventions make large programs much easier to understand, and you should follow them.

Give your methods meaningful names, and don't over-abbreviate words just to avoid typing a few keystrokes.

Method Body

The method header is followed by a body, which begins and ends with braces (curly brackets). One or more Java statements go inside the body. Put the opening brace at the end of the header line, and the closing brace directly under the first column of the header. Indent the lines within the body, relative to the header and the closing brace:

```
public void deposit() {
  statement1;
  statement2;
  ...
}
```

If the method header has any return type **except** void, then the method body must contain a statement made up of the keyword "return", followed by an expression of the same data type as the return type.

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

The header line indicates that this method returns a double. The body has a return statement, followed by an expression. Since the balance field is declared as a double, this expression results in a type double, so the program can successfully return the result of the expression.

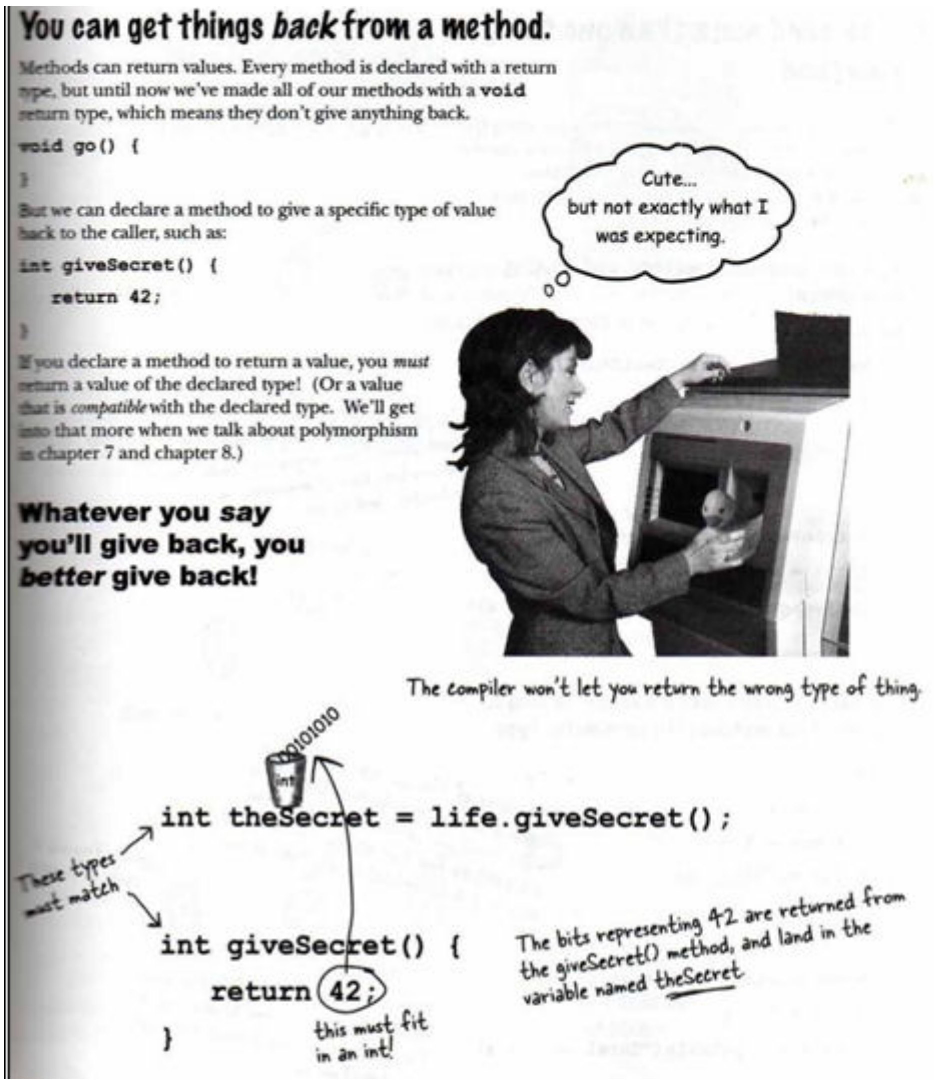
The method stops execution as soon as the return statement is executed, so it is usually the last statement in the method body.

It is a syntax error if the return type in the header is different from the data type in the return statement in the body. For example:

```
public int max(int num1, int num2) {
  double result;
  ...
  return result;  // a compile time error
}
```

The header says this method will return an int, but the return statement in the body returns a double. The compiler will not allow this since it would have to convert the double to an int to return it, and this could result in losing the decimals.

Here's another view of returning values:



You can get things *back* from a method.

Methods can return values. Every method is declared with a return type, but until now we've made all of our methods with a **`void`** return type, which means they don't give anything back.

```
void go() {
}
```

But we can declare a method to give a specific type of value back to the caller, such as:

```
int giveSecret() {
   return 42;
}
```

If you declare a method to return a value, you *must* return a value of the declared type! (Or a value that is *compatible* with the declared type. We'll get into that more when we talk about polymorphism in chapter 7 and chapter 8.)

Whatever you *say* you'll give back, you *better* give back!

```
int theSecret = life.giveSecret();

int giveSecret() {
    return 42;
}
```

Cute...
but not exactly what I
was expecting.
The compiler won't let you return the wrong type of thing.
01010101
int
These types must match
The bits representing 42 are returned from the giveSecret() method, and land in the variable named theSecret.
this must fit in an int!

Parameter list

The method name is always followed by parentheses (round brackets). Inside these parentheses, we declare variables. The variables declared in a method header are called *parameters*.

A parameter declaration is very similar to the declaration of a variable located **inside** the body of a method (called a *local variable*). We have already seen local variables declared in the methods of our programs. In the same way as a local variable, a parameter declaration is made up of a data type and a variable name. The rules for naming parameters are the same as for local variables: always start with a lower case letter. If the parameter name contains more than one word, each word except the first starts with a capital.

A method can have zero, one, or several parameters. Even if it has no parameters, the parentheses following the method name are still required.

Variable Scope and Lifetime

Variables can be declared outside a method - usually at the beginning of the class. This type of variable is called a **field/attribute** because it can be used by any method in the class (this is called the variable's **scope**). The variable is created when an object of the class is created and lives until the object dies (this is called the variable's **lifetime).**

A Java variable can also be declared inside a method – these are called **local variables**. Local means it can only be used within the method (scope). Local variables are created during the execution of the method (specifically, when the variable declaration is executed), and die when the method ends (lifetime). Local variables are not automatically initialized, and you should always assign them an initial value in the declaration statement.

*A variable's **scope** defines where in the program it can be used*.

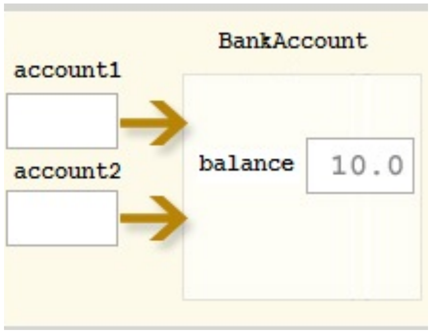
A *variable's **lifetime** is when it is created and when it is destroyed.*

Parameters are local variables as well and have same scope and lifetime as local variables.

Variables and Objects

In our code, we can have two reference variables refer to the same object (this code uses the BankAccount class we created earlier)

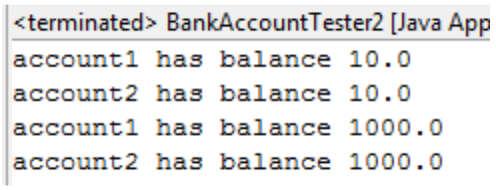
```
/* A program to show two variables referring to the same object.
*/
public class BankAccountTester2{
public static void main(String[]args){

// Declare variables to refer to BankAccount objects
BankAccount account1;
BankAccount account2;

// Create one BankAccount object and assign it to
// two variables.
account1=new BankAccount(1, 10.0);
account2=account1;
```



```
/* Print information about the BankAccount.  Using either variable name gives the same
result, since they both refer to the same object. */
System.out.println("account1 has balance "+account1.getBalance());
System.out.println("account2 has balance "+account2.getBalance());

// Change the size of the BankAccount.
```

```
account2.setbalance(1000.0);

/* Print information about the BankAccount.
   The object has been changed, so either variable shows the changed results. */
System.out.println("account1 has balance "+account1.getBalance());
System.out.println("account2 has balance "+account2.getBalance());
}
}
```

Here's the output from the program above:



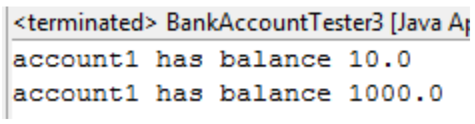
```
<terminated> BankAccountTester2 [Java App
account1 has balance 10.0
account2 has balance 10.0
account1 has balance 1000.0
account2 has balance 1000.0
```

Garbage Collection

Conversely, one reference variable can refer to two different objects (at different times, not simultaneously). Note that the first BankAccount object below becomes garbage once the second BankAccount object is created. Java's garbage collector looks after removing this object from memory for us:

```
/* A program to show one variable referring to two objects.
*/
public class BankAccountTester3{
public static void main(String[]args){
// Declare variable to refer to BankAccount objects
BankAccount account1;

// Step 1 - Create a BankAccount object and assign it to the variable.
account1 = new BankAccount(1, 10.0);

// Print information about the BankAccount.
System.out.println("account1 has balance "+account1.getBalance());

/* Step 2 - Create a new object and assign it to the variable. The old object becomes
garbage.  */
account1 = new BankAccount(2, 1000.0);

// Print information about the BankAccount.  The variable now refers to the new object.
System.out.println("account1 has balance "+account1.getBalance()));  } }
```

Here's the output from the program above:



```
<terminated> BankAccountTester3 [Java Ap
account1 has balance 10.0
account1 has balance 1000.0
```

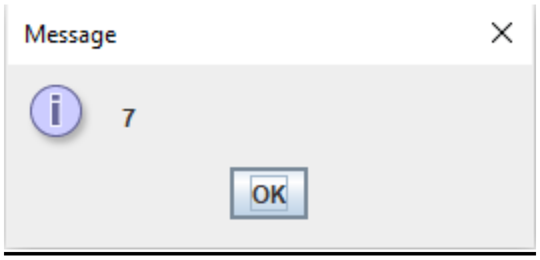
Message Dialog Boxes

We can fancy up our programs using Dialog Boxes for input and output rather than the console. To get dialog boxes, we use the Java Graphical User Interface, specifically the JOptionPane class. It's in the package javax.swing.JOptionPane, which we'll need to import.

We'll start with the easier ones. For output, we use MessageDialogs. As an example, using an integer variable called iNum which contains the value 7, we can display its value in a MessageDialog this way:

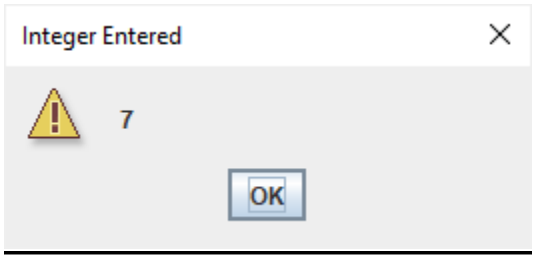
```
JOptionPane.showMessageDialog(null, iNum);
```

This produces a MessageDialog like the one below:



Note that the methods in the JOptionPane are all static, so we don't need an object of the JOptionPane class. We'll always set the first parameter to null since our programs are relatively simple and we aren't using parent and child components.

We can pretty it up with a title and a different icon to get the MessageDialog below:



```
JOptionPane.showMessageDialog(null, iNum, "Integer Entered", JOptionPane.WARNING_MESSAGE);
```

Icons will pop up for you to choose from once you type in the JOptionPane and the dot in the last parameter.

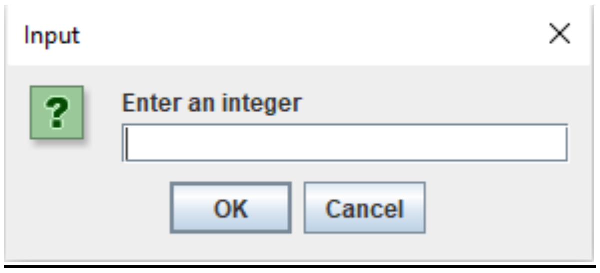
<u>Wrapper Classes and Parse Methods</u>

InputDialogs

For input, as an alternative to the console, we can use the InputDialog. For example:

```
String sNumber = JOptionPane.showInputDialog(null, "Enter an integer");
```

gives us this InputDialog that returns a String and assigns it to the variable sNumber:



Since we want an integer, we have to do some work to change it from a String to an int. We'll do this using Wrapper classes and Parse methods, which are described in detail below. Here is the statement we need: `intiNumber = Integer.parseInt(sNumber);`

Wrapper Classes and Parse Methods

The reason wrapper classes exist is to give us methods we can use with primitives data types, since primitives don't have methods. For our purposes, the main benefit of wrapper classes are their parse methods, which convert strings to other data types. This is called *parsing* the string. (The wrapper classes Boolean and Character for the primitive types boolean and char do not have parse methods.)

The methods are called parseByte, parseShort, etc. Each takes one argument of type String, and returns a value of the appropriate numeric type. The parse methods are static methods, so they are called by prefixing the method name with the name of the wrapper class to which it belongs (no object needed, just like the Math class and JOptionPane).