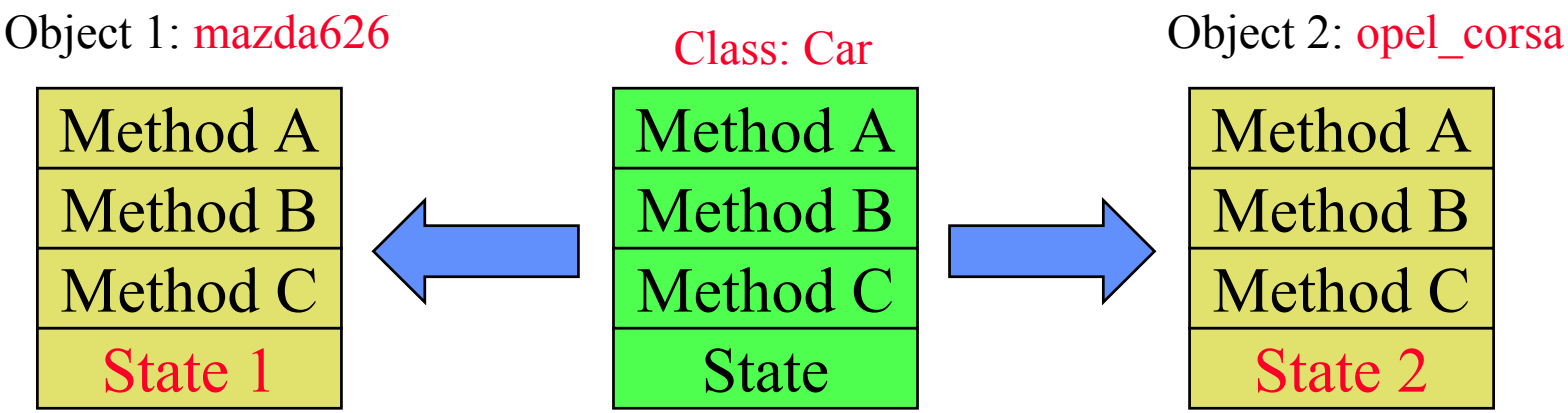
The Vocabulary of OOP

- **The most important term is `class`. A `class` is the template from which the `object` is actually made.**
- **To create an `object` you use the new keyword. It allocates memory and the built-in garbage collector will release memory when nobody uses the object anymore.**
- **When you create an `object` from a `class` , you are said to have created an `instance` of the class.**

```
Car mazda626 = new Car();
```

creates a new instance of the `Car` class.



- **Everything you write in Java is inside a class.**
- **Java is composed of many classes.**

The Vocabulary of OOP : Encapsulation

- **`Encapsulation` is another key concept of OOP. It means**
  - **combining** of **data** and **behaviour** in one **black box**
  - and **hiding** the implementation of the data from the user of the object
- **The data in an object are called `instance variables` or `fields`.**
- **Functions and procedures in a Java class are called its `methods`.**

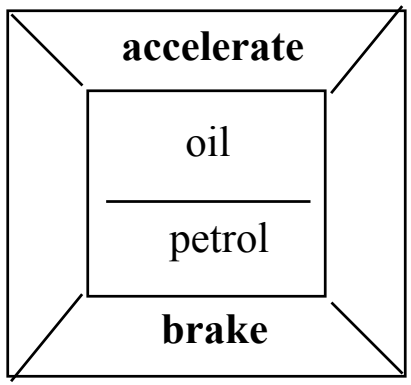


Figure: Encapsulation of data in an object

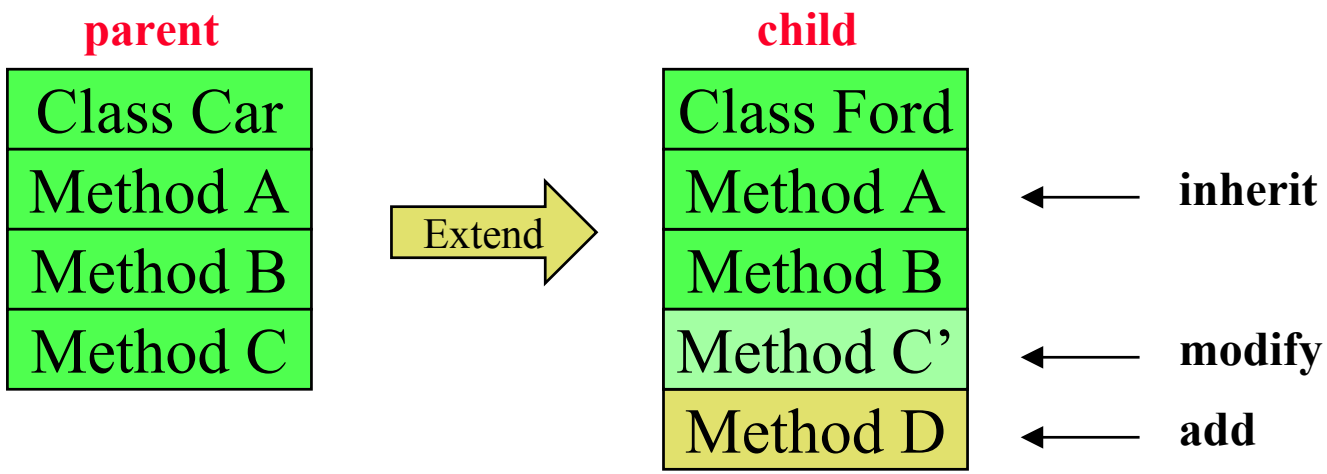
instance variables: oil, petrol (typically hidden from outside if declaired **private**)

methods: accelerate, brake (typically visible from outside if declaired **public**)

Encapsulation is the way to give the object its **"black box"** behaviour

The Vocabulary of OOP: Inheritance

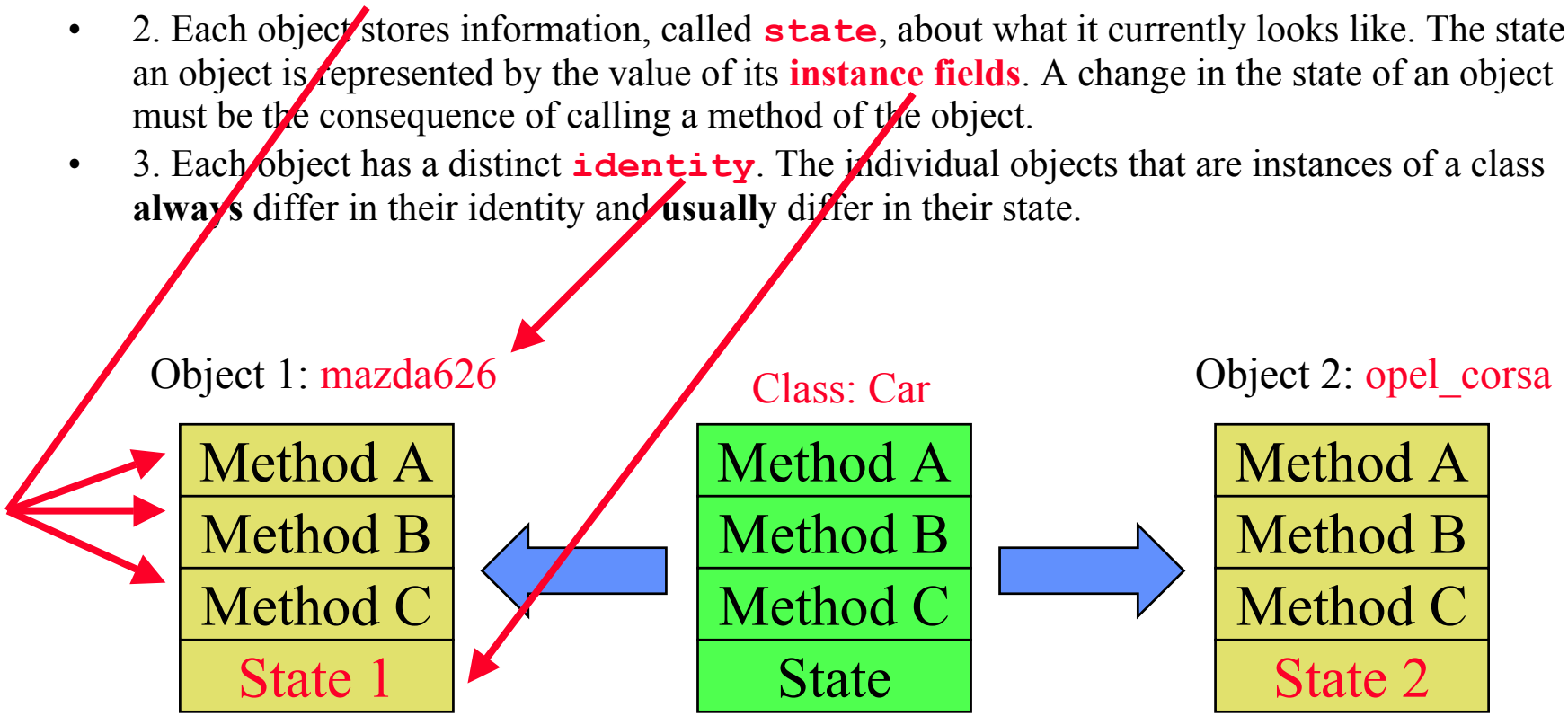
- **Classes can be built on other classes. We say that a class that builts on another class `extends` it.**
- **The general concept of extending a base class is called `inheritance`.**
- **When you extend a base class, the new class initially has all the properties and functions of its `parent`. You can choose whether you want to modify any function of the parent. You can also supply new functions that apply to the `child class` only. The same holds for the instance fields, too.**



- **In Java, all classes extend the "cosmic base class" called `Object`.**

Objects

- To work with OOP, you should identify **3 key characteristics** of objects:
  1. What is the object's **`behaviour`**? → **methods**
  2. What is the object's **`state`**? → **instance fields**
  3. What is the object's **`identity`**?
- 1. All objects that are instances of the same class share a family resemblance by supporting similar **`behaviour`**. The behaviour of an object is defined by the **methods** of its class.
- 2. Each object stores information, called **`state`**, about what it currently looks like. The state of an object is represented by the value of its **instance fields**. A change in the state of an object must be the consequence of calling a method of the object.
- 3. Each object has a distinct **`identity`**. The individual objects that are instances of a class **always** differ in their identity and **usually** differ in their state.



Relationship between classes

- The most common relationships between classes are:
  1. **use**
  2. **containment ("has-a")**
  3. **inheritance ("is-a")**

- **1. A class `uses` another class if it manipulates objects of that class. In general, a class `A` uses a class `B` if:**
  - **a/ a method of `A` sends a message to an object of class `B`, or**
  - **b/ a method of `A` creates, receives, or returns objects of class `B`.**

- **2. `Containment` means that objects of class `A` contain objects of class `B`. (Containment is a special case of use; if an `A` object contains a `B` object, then at least one method of class `A` will make use of that object of class `B`.**

- **The `inheritance` relationship denotes specialization. If class `A` extends class `B`, class `A` inherits methods from class `B`, but has more capabilities.**

- **Class diagrams** show the classes and their relationships.

Traditional versus OO programming

- In traditional structured programming

  algorithms come first

  data structures come second.

  First, you decide how to manipulate data; then you decide what structure to impose on the data.
- In object-oriented programming

  data structures come first

  algorithms come second.

  First, you create **abstract data structures**; then you look at the algorithms that operate on the abstract data structures.

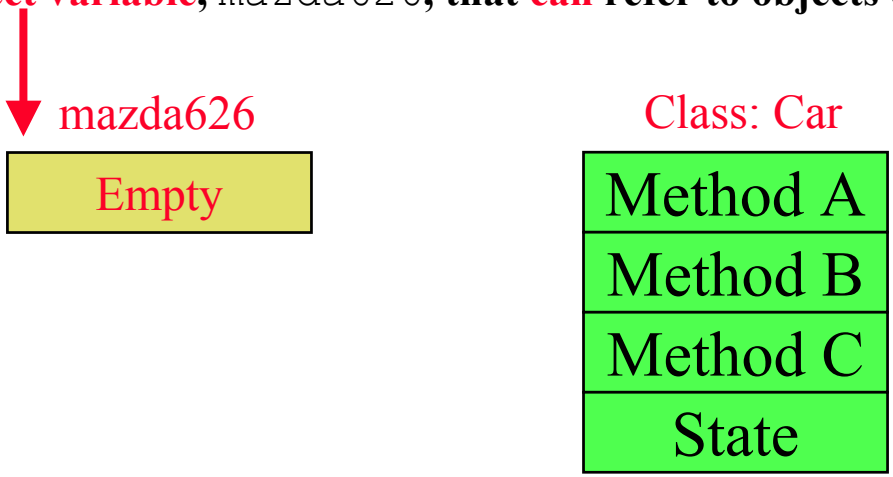
- In a traditional procedure-oriented program, you start the process at the **top**, with the **main program**.
- In object-oriented programming **there is no "top"**. You first find classes and then you add methods to each class. (A simple **rule of thumb** in identifying classes is to look for **nouns** in the problem analysis. Methods, on the other hand, correspond to **verbs**.)

Object Variables

- **For most classes in Java, you create objects, specify their initial state and then work with the objects.**
- **To access objects, you define object variables. The statement**

```
Car mazda626;          // mazda626 does not refer to any object
```

defines an object variable, `mazda626`**, that can refer to objects of type** `car`.



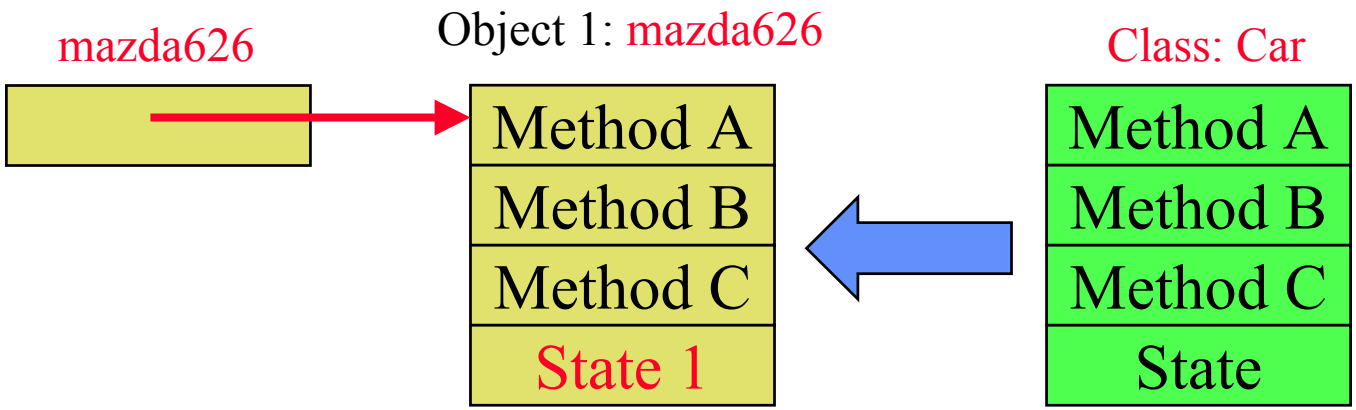
- The variable `mazda626` is **not an object and does not yet even refer to an object.** You cannot use any methods on the variable at this time.

```
mazda626.accelerate();          // not yet
```

Object Variables

- **Use the** new **operator to create an object:**

```
mazda626 = new Car();          // does create an instance
```



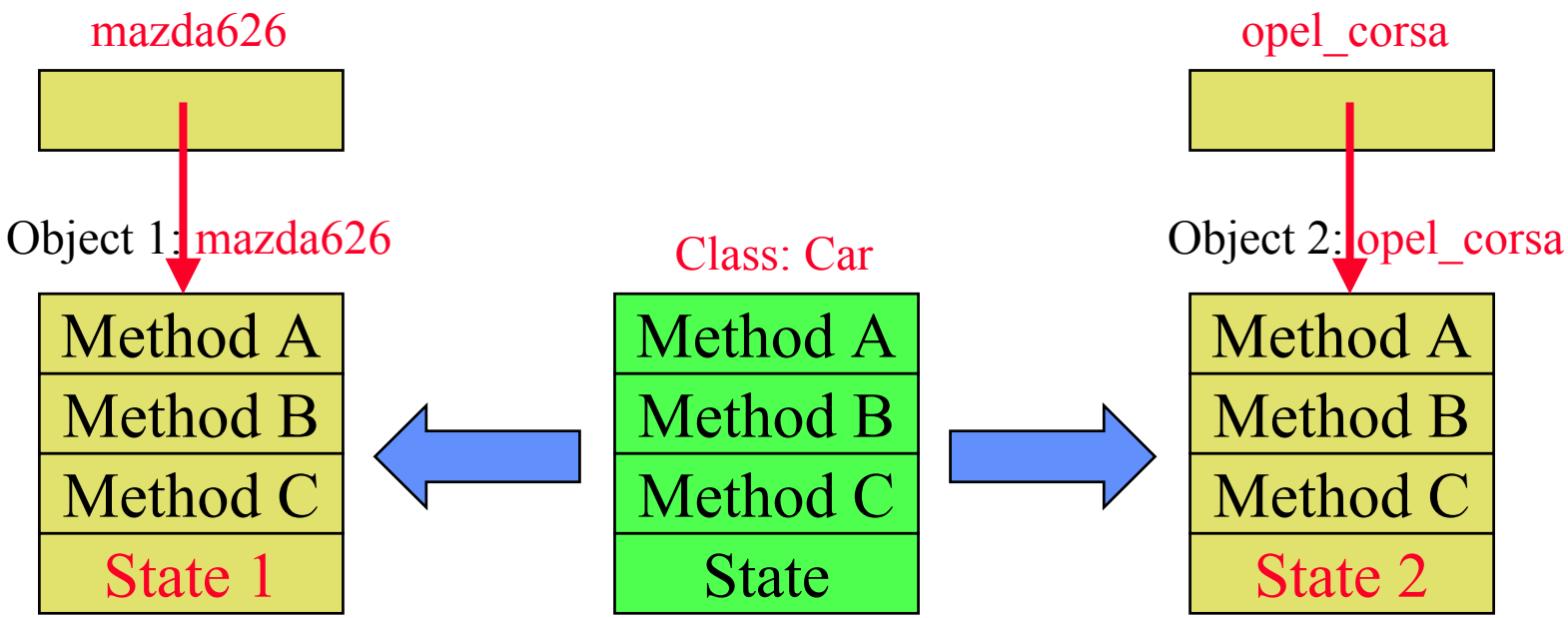
- **Now you can start to applying** `car` **methods to** `mazda626`**.**

Object Variables

- **You might need to create multiple objects (instances) of a single class:**

```
Car opel_corsa = new Car();
```

- **Now there are two objects of type `car`, one attached to the object variable `mazda626` and one to the object variable `opel_corsa`.**

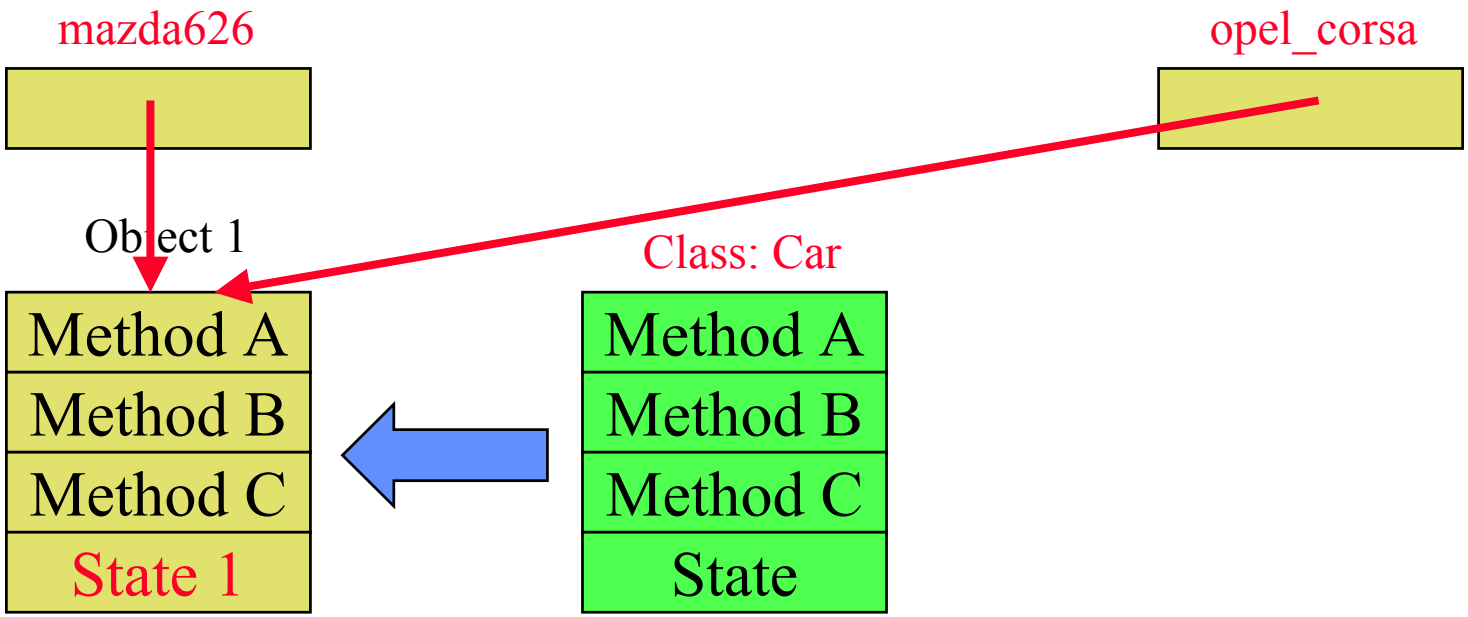


Object Variables (cont.)

- **If you assign one variable to another one using the equal sign,**

```
Car opel_corsa = mazda626;
```

then both variables refer to the same instance.



- **This can lead to surprising behaviour in your programs. For example, if you call**

```
mazda626.accelerate();
opel_corsa.brake();
```

the car object will accelerate and then brake, since the same **car object is referred to by** `opel_corsa` **and** `mazda626` **variables.**

Object Variables (cont.)

- **Many classes have a method called `clone` that makes a true copy. When you `clone` an existing object, you get a copy that reflects the current state of the object. Now the two objects exist independently, so they can diverge over time.**

- **You can explicitly set an object variable to `null` to indicate that it currently refers to no objects:**

```
opel_corsa = null;
if (opel_corsa != null) opel_corsa.brake();
```

- **If you call a method through a `null` variable, a run time error occurs.**

- **Object variables must be initialized either by**
  - **calling `new` or**
  - **by setting them to `null`.**

The Supplied `Date` Class

- **The `Date` class comes with Java. An instance of it has a state specifying the date and time:**

  Date todaysDate = new Date();

- **creates an instance of class `Date` and initializes its state to the current date (maintained by the operating system).**
- **You can also create an instance with a specific date:**

  ```
  Date preMillenium = new Date(99,12,31);
  ```

- **You can also set the time:**

  ```
  Date preMillenium = new Date(99,12,31,23,59,59);
  ```

- **Why is `Date` a class in Java rather than a built-in type, like `int`?**
- **By making `Date` into a class, the design task is off-loaded to a library designer. If the class is not perfect, other programmers can easily write their own `Date` class.**
- Notice that the **`Date`** class **has encapsulated data** to maintain the date. **It is irrelevant to know the internal representation of data inside the `Date` class.**

The Supplied `Date` Class (cont.)

- The **`Date`** class being in the **`java.util`** library has **25 methods**. Some of them are:

  Given a string representing a date and time, this method parses it and converts it to a time value.

  ```
  void parse(String s)
  ```

  This method returns true if the Date **comes before the date** when**.**

  ```
  boolean before(Date when)
  ```

  This method returns true if the `Date` **comes after the date** when**.**

  ```
  boolean after(Date when)
  ```

  This method converts the date held in the `Date` **object to a string representing the date using Unix date/time convention.**

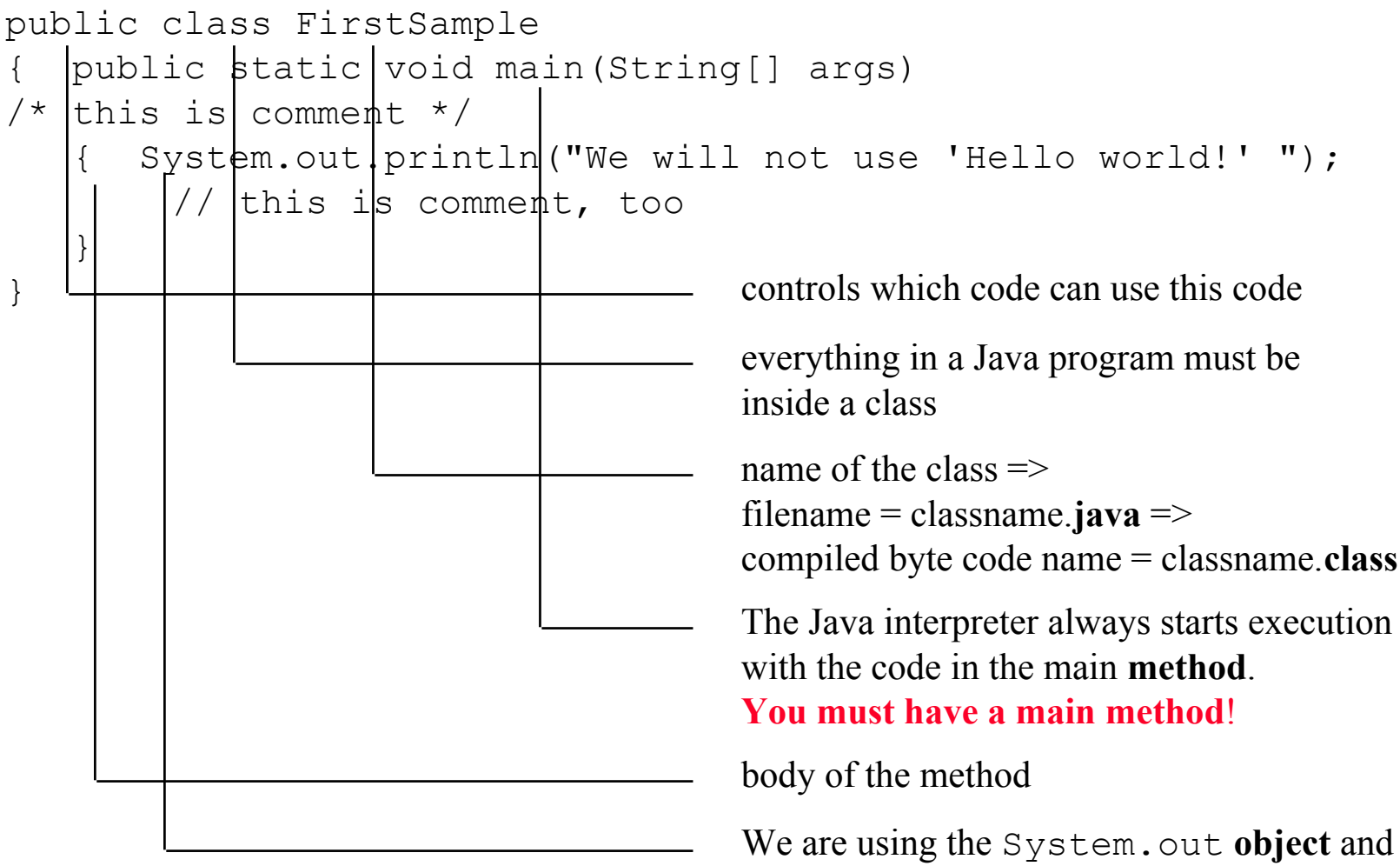
  ```
  string toString()
  ```

  This method converts the date held in the `Date` **object to a string representing the date using the local ordering convention.**

  ```
  string toLocaleString()
  ```

- **Exercise: Write a Java application to print out the current date in local ordering convention.**

A very simple Java Program



Exercise

- **Write a Java application to print out the current date in local ordering convention.**

```
import java.util.*;
public class WhatIsToday
{  public static void main(String arg[])
    {  Date today = new Date();
        System.out.println(today.toLocaleString());
    }
}
```

The Supplied Date Class (cont.)

- **Here is the list of the most important methods for getting at or changing the state of a `Date` instance:**

  Gets the day of the month of this date instance, a number between 1 and 31.

  ```
  int getDate()
  ```

  Gets the month of this date instance, a number between 0 and 11.

  ```
  int getMonth()
  ```

  Gets the year, with 0 denoting 1900, and so on.

  ```
  int getYear()
  ```

  Gets the weekday, a number between 0 and 6 (with 0 being Sunday).

  ```
  int getDay()
  ```

  Returns the hours, minutes, or seconds.

  ```
  int getHours(), int getMinutes(), int getSeconds()
  ```

  Sets the hours, minutes, or seconds.

  ```
  void setHours(int), void setMinutes(int), void setSeconds(int)
  ```

  Sets the current day of the month, the month and the year.

  ```
  void setDate(int), void setMonth(int), void setYear(int)
  ```

- **Convention in Java:** use **`get`** for **accessor methods** and **`set`** for **mutator methods.**

The Day Class

- The **`Day`** class being in the **`corejava.util`** package inside the **`\CoreJavaBook`** directory has the following methods:

  Advance the date currently set by a specified number of days.

  ```
  void advance(int n)
  ```

  Returns the day, month, or year of this day object. Days are between 1 and 31, months between 1 and 12, and years can be any year (such as 1996 or -333).

  ```
  int getDay(), int getMonth(), int getYear()
  ```

  Gets the weekday, a number between 0 and 6 (with 0 being Sunday).

  ```
  int weekday()
  ```

  This method is one of the main reasons to create the `Day` class. It calculates the number of days between the current instance of the `Day` class and instance b of the `Day` class.

  ```
  int DaysBetween(Day b)
  ```

  There are two ways (two constructors) to create an instance of the `Day` class:

  ```
  Day todaysDate = new Day();
  Day preMillenium = new Day(1999,12,31);
  ```

- **Exercise: Write a Java application to calculate how many days you have been alive.**

Exercise

Write a Java application to calculate how many days you have been alive.

```
import corejava.*;
public class DaysAlive
{  public static void main(String arg[])
 { int year;
   int month;
   int day;

   day = Console.readInt("Please, enter the day you were born.");
   month = Console.readInt("Please, enter the month you were born.");
   year = Console.readInt("Please, enter the year you were born.");

   Day today = new Day();
   Day birthday = new Day(year,month,day);
   System.out.println("You have been alive " +
                      today.daysBetween(birthday) + "days.");
 }
}
```

Exercise: `Calendar.java`

- **1. Study the application that prints out a calendar for the month and year specified in the command line argument.**
- **2. Run the application with commands:**

```
java Calendar
java Calendar 12 1999
```

- **3. To write such a calendar program you have to know how many days the month has. To solve the problem a Day object is created that starts with the first of the month:**

```
Day d = new Day(y, m, 1); // start date of the month
```

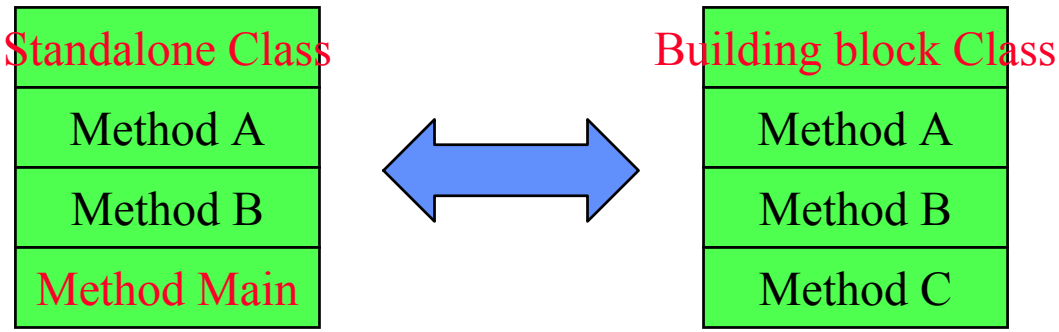
- **After printing each day, `d` is advanced by one day:**

```
d.advance(1);
```

- **When the month is advanced the program is stopped.**
- **Exercise: Modify the program to create a calendar for a whole year.**

Starting to Build Your Own Classes

- So far we have written classes to run as **stand-alone programs**. In these classes the Java interpreter looked for the **main** method and ran it. The **main** method called other methods of the class as needed.



- Now we want to write classes that do not stand alone, rather they are the **building blocks** for **constructing** stand-alone programs.
- **The simplest syntax for a class in Java:**

```
class NameOfClass
{  // definitions of the class's features
   // includes methods and instance fields
}
```

- **The outermost pair of braces (block) defines the code that will make up the class.**
- **Our convention:** to use initial caps for class names.
- **Example: an** `Employee` **Class** that might be used by a business in writing a payroll system.

```
class Employee
{  public Employee(String n, double s, Day d)
   {  name = n;
      salary = s;
      hireDay = d;
   }
   public void print()
   {  System.out.println(name + " " + salary + " " + hireYear());
   }
   public void raiseSalary(double byPercent)
   {  salary *= 1 + byPercent / 100;
   }
   public int hireYear()
   {  return hireDay.getYear();
   }
   private String name;
   private double salary;
   private Day hireDay;
}
```

Analyzing the `Employee` Class

- **The `Employee` class has four methods:**

```
public Employee(String n, double s, Day d)
public void print()
public void raiseSalary(double byPercent)
public int hireYear()
```

- The keyword **`public`** is usually called as **access modifier**. In Java, these **access modifiers** describe who can use the method or who can use the class if a modifier is used in the name of the class.
- The keyword **`public`** means that **any method in any class** that has access to an instance of the `Employee` class can call the method.
- There are **four possible access levels** as explained later**.**
- There are three **instance fields** that hold the data to be manipulated inside an instance of the Employee class**:**

```
private String name;
private double salary;
private Day hireDay;     // instance of the Day class
```

- The keyword **`private`** makes sure that **no outside object can access** the instance fields except the methods of our class.

First Step with Constructors

- **Let's look at the first method listed in our** Employee **class:**

```
public Employee(String n, double s, Day d)
{ name = n;
  salary = s;
  hireDay = d;
}
```

- This is an example of a ***constructor method***. It is used to construct an object from the class by **initializing the instance variables.**
- **For example, you create an instance of the** Employee **class with code like this:**

```
hireDate = new Day(1950,1,1);
Employee number007 = new Employee("James Bond",100000,hireDate);
```

- **The *constructor method* is called when the class is created by `new` and the *constructor method* initializes the instance fields.** In the example above:

```
name = "James Bond";
salary = 100000;
hireDay = January 1, 1950;
```

First Step with Constructors (Cont.)

- The **new** method is always used together with a **constructor** to create the class. This forces you to set the initial state of your objects. In Java, **you cannot create an instance without initialization**.
- **Rules of using constructors:**
  1. **A constructor has the same name as the class.**
  2. **A constructor may take one or more (or even no) parameters.**
  3. **A constructor is always called with the new keyword.**
  4. **You can't apply a constructor to an existing object to reset the instance fields.**

     Of course, if resetting all fields of a class is important, the class designer can provide a **mutator method** such as **`empty`** or **`reset`** for that purpose.
- **It is possible to have more than one constructor in a class. You have already seen it in the** Day **class:**

```
Day todaysDate = new Day();
Day preMillenium = new Day(1999,12,31);
```

The Methods of the `Employee` Class

- **Methods can access the private instance fields by name and can modify their values (**these are the ***mutator methods*****).** An example of that is the **`raiseSalary`** method:

```
public void raiseSalary(double byPercent)
{  salary *= 1 + byPercent/100;
}
```

- **void** means that this method does not return any value.
- The most interesting method is **`hireYear`**:

```
public int hireYear()
{  return hireDay.getYear();
}
```

- This method returns an integer value, and it does this by **applying a method to the `hireDay` instance variable**. Indeed, `hireDay` is an instance of the `Day` class, which has a `getYear` method.
- Finally, the `print` method is an example of an ***accessor method*****:** It simply accesses (prints out) the current state of the instance variables:

```
public void print()
{System.out.println(name + " " + salary + " " + hireYear());
}
```

Use of the `Employee` Class

- **Exercise: study the `Employee` Class and run the `EmployeeTest` program.**

```
import java.util.*;
import corejava.*;
public class EmployeeTest
{  public static void main(String[] args)
   {  Employee[] staff = new Employee[3];
      staff[0] = new Employee("Harry Hacker", 35000,
         new Day(1989,10,1));
      staff[1] = new Employee("Carl Cracker", 75000,
         new Day(1987,12,15));
      staff[2] = new Employee("Tony Tester", 38000,
         new Day(1990,3,15));
      int i;
      for (i = 0; i < 3; i++) staff[i].raiseSalary(5);
      for (i = 0; i < 3; i++) staff[i].print();
   }
}
```

Summary of method types

- When the user of a class has a legitimate interest in both reading and writing an instance field, the **class implementor should supply three items**:

  1. **A private data field.**
  2. **A public accessor method to access the private data field.**
  3. **A public mutator method to modify the private data field.**

- **This approach has the benefits:**

  1. **The internal implementation can be changed without affecting any code outside the class.**
  2. **Mutator methods can perform *error-checking*, whereas code that simply assigns to a field cannot. For example, use**

```
setDate(1999,3,31);    //with internal error-checking
```

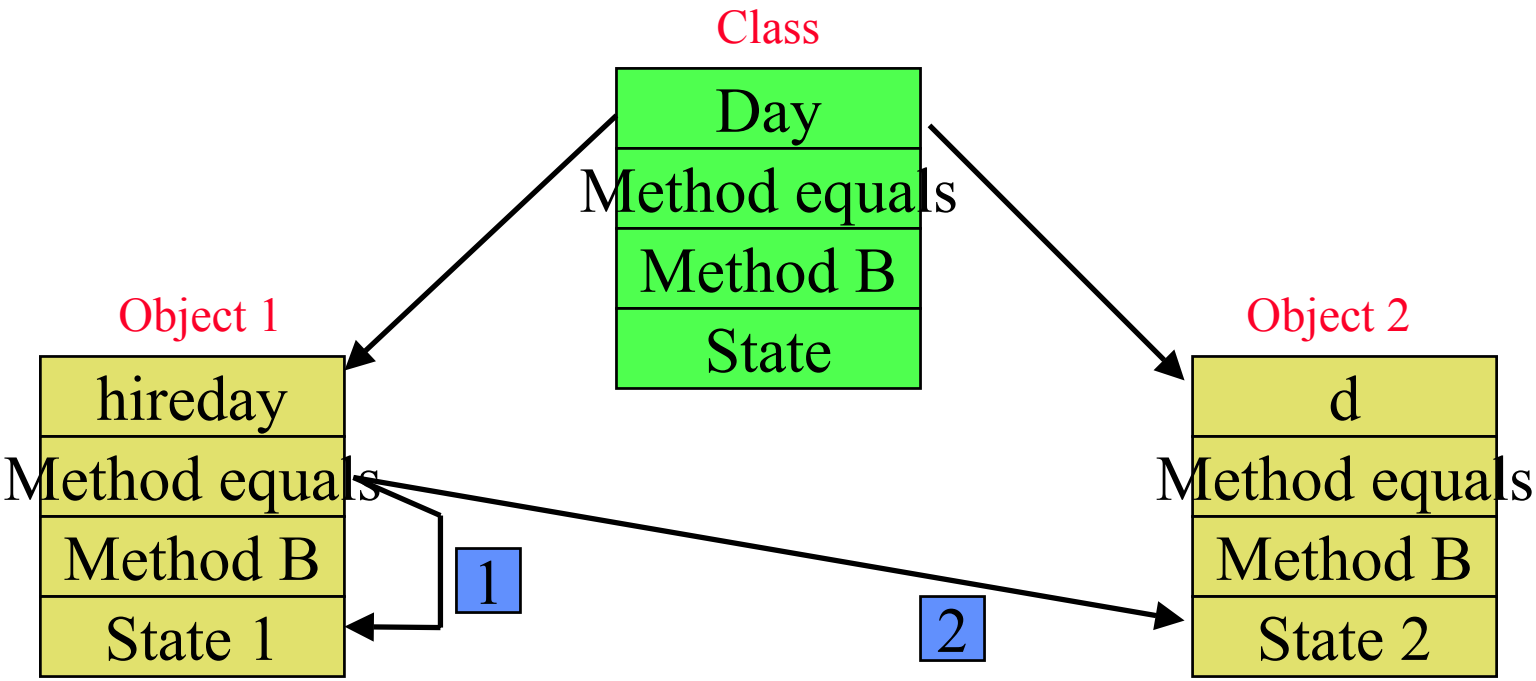
instead of

```
d.setDay(31);          // problem in case of February
d.setMonth(2);
```

Method Access to Private Data

- **There are two rules of method accesses to private data (state of object):**

  1. A method can access the private data of **the object on which it is invoked.**



  2. **Surprise**: A method can access the private data of **all sibling objects in its class!**

Method Access to Private Data

- **There are two rules of method accesses to private data:**

  1. A method can access the private data of **the object on which it is invoked.**
  2. A method can access the private data of **all objects of its class.**

- **For example, consider the method** `daysBetween` **and its usage in the** `DaysAlive` **class:**

```
Day today = new Day();
Day birthday = new Day(year,month,day);
System.out.println("You have been alive " +
                   today.daysBetween(birthday) + "days.");
```

The method `daysBetween` **accesses the private fields of** `today` **(rule 1). It also accesses the private fields of** `birthday` **(rule 2). This is legal because** `birthday` **is a sibling object of** `today`**.**

Class Variables

- **A *class variable* can be accessible by all the methods in the class.** They are declared **before** the **main** method using the following syntax:

```
class Employee
{  private static double socialSecurityRate = 7.62;
   public static void main(String[] args)
   { . . . }
}
```

- **By replacing the keyword `private` with the keyword `public` one can create true global variables accessible by all methods in an application:**

```
class Employee
{  public static double socialSecurityRate = 7.62;
   public static void main(String[] args)
   { . . . }
}
```

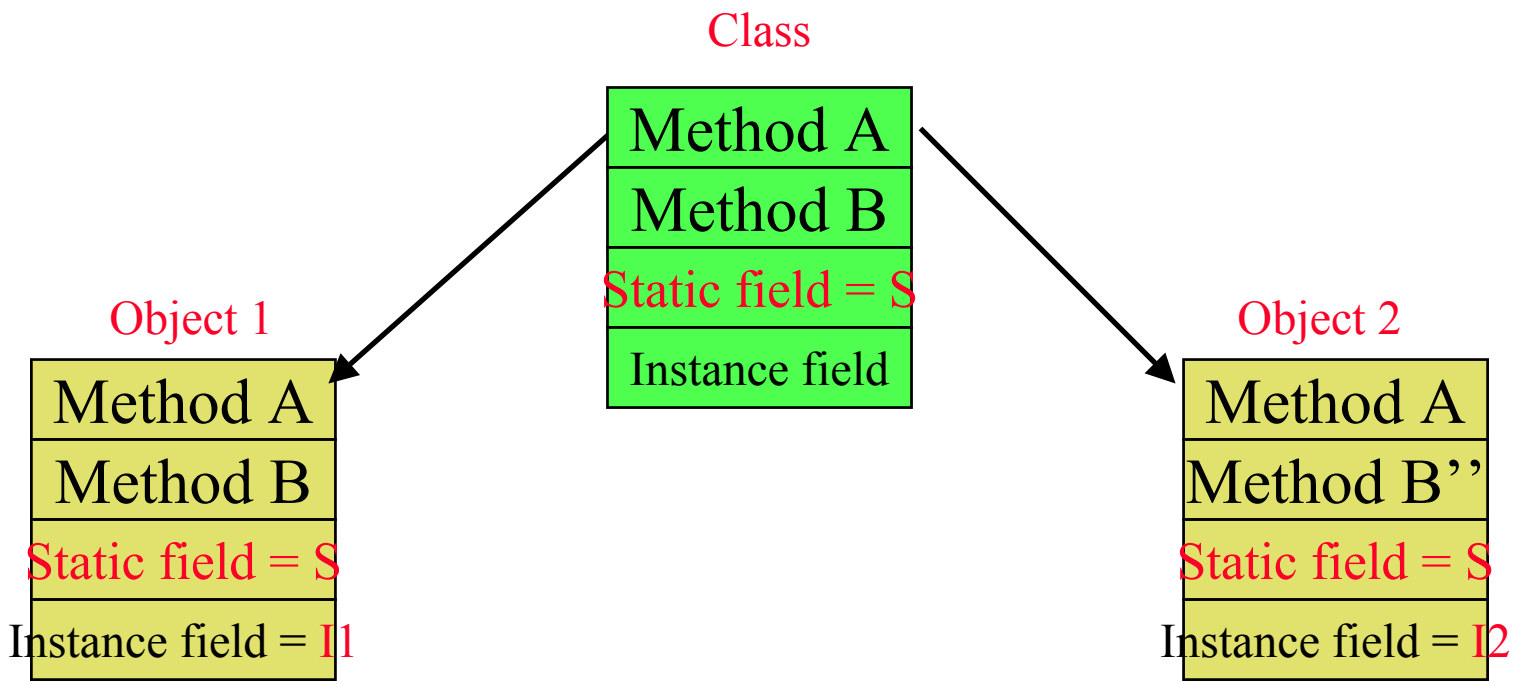
DO NOT USE GLOBAL VARIABLES!

Pivate Methods

- When implementing a class, **we make all data fields private**, but what about the methods?
- **Private methods** are quite frequent, and they can be called only from other operations of the same class.
- To implement a private method in Java, simply change the **`public`** keyword to **`private`**.
- **In sum, choose private methods:**
  1. **for those functions that are of no concern to the class user and**
  2. **for those functions that could not easily be supported if the class implementation were to change.**

Static Fields (Variables)

- Classes can have both **static fields** and **instance fields.** Use the **static** keyword to make a field static. Other fields are instance fields and they represent the state of the objects, i.e. they can be different for different objects.
- **Static fields (variables) do not change from one instance of a class to another**, so you should think of them as belonging to a class.



Static Methods

- Classes can have both **static variables** and **static methods.**

- **Static methods belong to a class and do not operate on any instance of a class**. It means **you can use them without creating an instance (object) of a class.** The static methods are staticly available without dynamic object creation.

- For example, all of the methods in the Console class are static methods. This is why a syntax like

  ```
  x = Console.readDouble();
  ```

  makes perfect sense without using `new` to create a console object.

- **The general syntax for using a static method from a class is:**

  ```
  ClassName.staticMethod(parameters);
  ```

- **WARNING: Because static methods can work without an object of the class they can only access static fields.**

Static Methods as Headers

- **Consider the header for the `main` method:**

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

- **Since `main` is `static`, you don't need to create an instance of the class in order to call it - and the Java interpreter doesn't either.**

- **For example, if your `main` function is contained in the class `Mortgage` and you start the Java interpreter with:**

```
Java Mortgage
```

then the interpreter simply starts the `main` function without creating an object of the `Mortgage` class.

RandomIntGenerator.java

- **Java provides a random number generator. The random number is supplied by a call to:**

```
java.lang.Math.random();
```

- **We create a better random number generator with the following advantages:**

  1. Adds the convenience of generating random integers in a specific range.

  2. It is more "random" than the one supplied with Java.

- **The code is shown below:**

```
public class RandomIntGenerator
{  /*   @param l the lowest integer in the range
         @param h the highest integer in the range
         Used to return a random integer in the range */
public RandomIntGenerator(int l, int h)  // a constructor
{  low = l;
   high = h;
}
```

Static initialization block

- **Data structures of the class** `RandomIntGenerator`**:**

```
private static final int BUFFER_SIZE = 101;
private static double[] buffer = new double[BUFFER_SIZE];
static /* initialization block */
{  int i;
        for (i = 0; i < BUFFER_SIZE; i++)
        buffer[i] = java.lang.Math.random();
}
/* instance fields: */
private int low;
private int high;
```

- It uses a small static array which is filled up with a **static initialization block.** Use these blocks whenever simple initialization statements for static members are either not possible or too clumsy. In the example we need a loop to initialize the buffer array, and **a loop cannot be coded with a simple initializer.**
- **The syntax for a static initialization block is simply the keyword static followed by braces that mark any Java code block.**
- **Java then executes the block before any method of the class is called.**
- **You can have many such blocks in a class.**

RandomIntGenerator.java

- **It has a public method, called `draw`, for drawing a random integer in the specified range.**

```
public int draw()
{  int r = low
        + (int)((high - low + 1) * nextRandom());
   if (r > high) r = high;
   return r;
}
```

- **The `draw` method uses a static method called `nextRandom` that actually implements the algorithm. It uses the static `buffer` array (static method can access only static fields) and calls the Java built-in random number generator twice:**

```
private static double nextRandom()
{  int pos =
        (int)(java.lang.Math.random() * BUFFER_SIZE);
   if (pos == BUFFER_SIZE) pos = BUFFER_SIZE - 1;
   double r = buffer[pos];
   buffer[pos] = java.lang.Math.random();
   return r;
}
```

- **Notice that the static keyword can be omitted in the definition of `nextRandom` without any problem, i.e., it could be an ordinary method.**

Overloading

- **It is possible to have more than one constructor in a class. You have already seen it in the `Day` class:**

```
Day todaysDate = new Day();
Day preMillenium = new Day(1999,12,31);
```

- This capability is called **overloading.**
- **Overloading** occurs if several methods have the **same name** but **different arguments.**
- The Java interpreter has to sort out which method to call. A **compile-time error** occurs if the compiler cannot match the arguments or if more than one match is possible.
- **WARNING: method overloading** (sometimes called **ad-hoc polymorphism**) must be distinguished from **true polymorphism**, which Java also does support**.**

Instance Field Initialization

- Instance field initialization is done by **constructors**.
- If your class has no constructors, Java provides a **default constructor** for you. A **default constructor** is a constructor with no parameters. **It sets all the instance variables to a default value** (numbers to zero, objects to null).
- **This only applies when your class has no constructors.**
- **Warning: Instance variables differ from local variables in a method. Local variables must be always initilized explicitly.**
- **If all constructors of a class need to set a particular instance variable to the same value, there is a convenient syntax for doing the initialization.** For example, **nextOrder** to **1**:

```
class Customer
{   public Customer(String n)
    {     name = n;
          accountNumber = Account.getNewNumber(); }
    public Customer(String n, int a)
    {     name = n;
          accountNumber = a; }
    private String name;
    private int accountNumber;
    private int nextOrder = 1;
}
```

The "this" Object

- In a method, the **keyword `this`** refers to the object (in its entirety) on which the method operates.
- Many Java classes (for example, **`date`**) have a method called **`toString()`** that prints out the object. You can print out the current date by saying:

```
this.toString();
```

- **More generally, provided your class implements a** toString() **method, you can print it out symply by calling:**

```
System.out.println("Customer: " + this);
```

- **This is a useful strategy for debugging.**
- **If the first line of a constructor has the form `this(...)`, then the constructor calls another constructor of the same class:**

```
class Customer
{   public Customer(String n)
    {     this(n, Account.getNewNumber()); }
    public Customer(String n, int a)
    {     name = n;
          accountNumber = a; }
...}
```

Exercise: CardDeck.java

- **A simple card game: the program chooses two cards at random, one for you and one for the computer. The highest card win. It repeats 10 times and then prints the result.**
- **Exercise:**
  1. **Run the code of CardDeck.java**
  2. **Study the code of CardDeck.java**
  3. **Study the code of Card.java**
  4. **Is there any method in Card.java which is not used?**
     (yes getValue, getSuit)
  5. **Remove the superfluous method(s) of Card.java , compile it and run the code of CardDeck.java again.**

Class Design Hints

1. **Always keep data private**
2. **Always initialise data (Java won't initialise local variables for you)**
3. **Don't use too many basic types in a class**

   The idea is to replace multiple **related** uses of basic types with other classes. For example, replace the following instance fields in a `Customer` class

   ```
   private String street;
   private String city;
   private String state;
   ```

   with a new class called `address`.
4. **Not all fields need individual field accessors and mutators**
5. **Break up classes with too many responsibilities**
6. **Make the names of your classes and methods reflect their responsibilities**

Class Design Hints (cont.)

7. Use a standard form for class definitions

- **public features**
- **package scope features**
- **private features**

Within each section, we list

- **constants**
- **constructors**
- **methods**
- **static methods**
- **instance variables**
- **static variables**