Programmierpraktikum

Claudius Gros, SS2012

Institut für theoretische Physik
Goethe-University Frankfurt a.M.

Object-Oriented Programming (OOP)

object-oriented programming

principles

- **modularity**
  The source code for an object can be written and maintained independently and used by many distinct applications.
- **information hiding**
  The users needs to know only about the functionality of the public methods, the complexity of the internal details are hidden to the outside world (data abstraction).
- **inheritance**
  A class may inherite properties from other classes making it easy to define subytpes (vehicle → car → cabriolet).
- **polymorphism**
  A class may come with a range of distinct constructors and member functions implementing semantically related task.

```
1. static double   abs(double a);
2. static int      abs(int a);
3. static long     abs(long a);
```

- object oriented programming tries to cast programming into structure resembling human though pathways

classes and objects

- in Java objects are classes
- classes can be defined in the same file as the executable class containing the **`main()`** function or in seperated files
- classes may
  - *extend another class*
  - *implement various interfaces*
  - *have user defined construtors*
- classes need to instantiated (with **`new`**) to create a modifiable instance

```
1. public class MyClass {
2.
3. int myVariable1;         // member variables can be
4. double[] myVariable2;    // private, public, static, ...
5.
6. /** Define any number of member functions (methods).
7.   * They can have as return values any primitive data type or
```

```
 8.   * any predefined or user defined object, or nothing (void).
 9.   */
10.   void myFunction1 () {
11.   } // end of MyClass.myFunction1()
12.
13. }   // end of class MyClass
```

object definition example: BookBasics

- **private** variables (functions) can be accessed only within the defining class
  it is custom to use standardized public functions to modify private variables (getter/setter)
- **public** variables (functions) can be accessed by other classes
- contains no **main()** method $\rightarrow$ not executable by itself

```
 1. public class BookBasic {
 2.
 3.   private int itemCode; // Internal variables.
 4.   private String title;
 5.
 6. /* Standard getter for itemCode.
 7.  */
 8.   public int getItemCode() {
 9.     return itemCode;
10.   }
```

```
11.
12. /* Standard setter for itemCode, returns 'true' if
13.  * itemCode is valid, otherwise 'false'.
14.  */
15.   public boolean setItemCode (int newItemCode) {
16.     if (newItemCode > 0) {      // a given constraint
17.       itemCode = newItemCode;
18.       return true;
19.     } else
20.       return false;
21.   }
22.
23. /* Standard getter for title.
24.  */
25.   public String getTitle() {
26.     return title;
27.   }
28.
29. /* Standard setter for title.
30.  */
31.   public void setTitle (String newTitle) {
32.     title = newTitle;
33.   }
```

```
34.
35. /* Non-trivial member function.
36.  */
37.   public void display() {
38.     System.out.println(itemCode + " " + title);
39.   }
40. } // end of class BookBasic
```

object definition usuage: BookBasics

- access member functions inside residing class:
  functionName(*args*) or
  this.functionName(*args*) if function overloads an inherited function
- access member functions from the outside:
  name.functionName(*args*) where
  name is the name of the instantiated class object

```
 1. public class UseBookBasic {
 2.
 3.   public static void main(String[] args) {
 4.     BookBasic b = new BookBasic();      // instantiation of a book
 5.     b.setItemCode(5011);                // code of book
 6.     b.setTitle("Fishing Explained");    // book title
 7.
 8. // --- printing by hand and using the printing routine of BookBasic
 9.     System.out.println(b.getItemCode() + " " + b.getTitle());
10.     System.out.print("From BookBasic.display(): ");
11.     b.display();
```

```
12.   }
13. }
14.
15. // *** *************** ***
16. // *** class BookBasic ***
17. // *** *************** ***
18.
19. class BookBasic {
20.
21.   private int itemCode;
22.   private String title;
23.
24.   public int getItemCode() { return itemCode; }
25.
26.   public boolean setItemCode (int newItemCode) {
27.     if (newItemCode > 0) {
28.       itemCode = newItemCode;
29.       return true;
30.     } else
31.       return false;
32.   }
33.
34.   public String getTitle() { return title; }
```

```
35.
36.   public void setTitle (String newTitle) {
37.     title = newTitle;
38.   }
39.
40.   public void display() {
41.     System.out.println(itemCode + " " + title);
42.   }
43. }
```

Java access modifiers

- **public**
  A *public* class or member function can be accessed by other classes.
  Filename must match class name.
- **private**
  A *private* variable or member function can be accessed only inside the residing class.
- **abstract**
  An *abstract* class may not be instantiated, only respective subclasse may (similar to *interfaces*).
- **final**
  Final classes / methods / variables
  cannot be extended / overwritten / changed.
- **static**
  A *static* class always exists, even if not instantiated, and can be accessed

directly.

```
1. double x = Math.random();  // the class Math is static and
2. double y = Math.sqrt(x);   // needs not to be instantiated
```

class constructors

- the constructor specifies what to do when instantiating a class, it has per definition the name of the residing class
- the constructor function has no return value

```
 1. public class BookWithConstructor {
 2.
 3.   private int itemCode;
 4.   private String title;
 5.
 6. /* User defined constructor.
 7.  */
 8.   public BookWithConstructor(int newItemCode, String newTitle) {
 9.     setItemCode(newItemCode);   // use always the setter if existing,
10.     setTitle(newTitle);         // there might be contraints
11.   }
12.
13.   public int getItemCode() {
14.     return itemCode;
```

```
15.   }
16.   public boolean setItemCode (int newItemCode) {
17.     if (newItemCode > 0) {
18.       itemCode = newItemCode;
19.       return true;
20.     } else
21.       return false;
22.   }
23.   public String getTitle() {
24.     return title;
25.   }
26.   public void setTitle (String newTitle) {
27.     title = newTitle;
28.   }
29.   public void display() {
30.     System.out.println(itemCode + " " + title);
31.   }
32. }
```

class instantiation

- with **new** a new instantiation (realization) of a class is created
- **new ... ()** calls the class constructor
- there is no limit on the number of instantiations

```
 1. public class UseBookWithConstructor {
 2.
 3.   public static void main(String[] args) {
 4.     BookWithConstructor b =
 5.       new BookWithConstructor(5011, "Fishing Explained");
 6.     b.display();
 7.   }
 8. }
 9.
10. // *********************************
11. // *** class BookWithConstructor ***
12. // *********************************
13.
14. class BookWithConstructor {
```

```
15.
16.   private int itemCode;
17.   private String title;
18.
19. /* User defined constructor.
20.  */
21.   public BookWithConstructor(int newItemCode, String newTitle) {
22.     setItemCode(newItemCode);
23.     setTitle(newTitle);
24.   }
25.
26.   public int getItemCode() {
27.     return itemCode;
28.   }
29.   public boolean setItemCode (int newItemCode) {
30.     if (newItemCode > 0) {
31.       itemCode = newItemCode;
32.       return true;
33.     } else
34.       return false;
35.   }
36.   public String getTitle() {
37.     return title;
```

```
38.   }
39.   public void setTitle (String newTitle) {
40.     title = newTitle;
41.   }
42.   public void display() {
43.     System.out.println(itemCode + " " + title);
44.   }
45. }
```

method overloading (polymorphism)

- methods may be overloaded when differing by argument type (or number) and/or return value

```
 1. public class BookMultiConstructor {
 2.   private int itemCode;
 3.   private String title;
 4.
 5. /* The first constructor.
 6.  */
 7.   public BookMultiConstructor(int newItemCode, String newTitle) {
 8.     setItemCode(newItemCode);
 9.     setTitle(newTitle);
10.   }
11.
12. /* The second constructor.
13.  */
14.   public BookMultiConstructor(String newTitle) {
15.     setItemCode(0);
16.     setTitle(newTitle);
```

```
17.   }
18.
19.   public int getItemCode() {
20.     return itemCode;
21.   }
22.   public boolean setItemCode (int newItemCode) {
23.     if (newItemCode > 0) {
24.       itemCode = newItemCode;
25.       return true;
26.     } else
27.       return false;
28.   }
29.   public String getTitle() {
30.     return title;
31.   }
32.   public void setTitle (String newTitle) {
33.     title = newTitle;
34.   }
35.   public void display() {
36.     System.out.println(itemCode + " " + title);
37.   }
38. }
```

- keep **`class`** *`BookMultiConstructor`* as seperated class file
- you need to compile only the class containing the **`.main()`** method
  all other classed needed are automatically included from current path

```
 1. public class UseBookMultiConstructor {
 2.   public static void main(String[] args) {
 3.
 4.     BookMultiConstructor b =
 5.        new BookMultiConstructor(5011, "Fishing Explained");
 6.     b.display();
 7.     // note the immediate call to a method on a new instance
 8.     new BookMultiConstructor("Dogs I've Known").display();
 9.   }
10. }
```

static functions

- *abstract* classes with *static* functions
  are convenient for general purpose utilities
- any class can contain *static* functions
- return types of member function can be a primitive data types,
  an objects (class) or void:
  boolean, *int[]*, *void*, ...

```
 1. public class MyUtilityDemo {
 2.
 3. public static void main(String[] args)
 4.   {
 5.   double rr = 11.123456;
 6.   MyUtility.printString("initial rr: " + String.valueOf(rr));
 7.   rr = MyUtility.roundToDigits(rr, 2);
 8.   MyUtility.printString("  2 digits: " + String.valueOf(rr));
 9.   }  // end of MyUtilityDemo.main()
10.
```

```
11. }    // end of class MyUtilityDemo
12.
13. // *** *************** ***
14. // *** class MyUtility ***
15. // *** *************** ***
16.
17. /** Example of a abstract (cannot be instantiated) class.
18.   * An abstract class can have only static member functions.
19.   */
20. abstract class MyUtility {
21.
22. /** Just printing a String to standard ouput.
23.   * Can be generalized to print (append) data to a file.
24.   */
25. static void printString(String output)
26.  {
27.  System.out.println("MyUtility.printString():: " + output);
28.  }
29.
30. /** Rounds to n digits accuracy.
31.   */
32. static double roundToDigits(double input, int nDigits)
33.  {
```

```
34.  for (int i=0; i<nDigits; i++)
35.    input *= 10.0;
36.  input = Math.round(input);
37.  for (int i=0; i<nDigits; i++)
38.    input *= 0.1;
39.  return input;
40.  }
41.
42. }   // end of class MyUtility
```

extending and inheriting

- hierarchical object definition possible
- a **`class`** which *extends* another **`class`** inherits all its member functions and variables
- an **`interface`** is implemented and not extended
- a **`class`** can extend only one other class but can implement many interfaces
- the parent (root) of all classes is the **`Object`** class

```
1. public class ExtendingDemo {
2.
3. public static void main(String[] args) {
4.
5.   System.out.println(" ");
6.
7.   ExtendingSimpleClass esc = new ExtendingSimpleClass(3);
8.   System.out.printf("calling ExtendingSimpleClass.getTopSecret()  : %d\n",
```

```
 9.                     esc.getTopSecret()); // inherited from SimpleClass
10.   System.out.printf("calling ExtendingSimpleClass.secretPrinting(): ");
11.                     esc.secretPrinting();
12.
13. // --- an ExtendingSimpleClass is a SimpleClass
14.   SimpleClass sc = new ExtendingSimpleClass(4);
15.   int dummy = sc.getTopSecret();              // possible
16. //            sc.secretPrinting();            // error
17.   ((ExtendingSimpleClass)sc).secretPrinting(); // possible: casting beforehand to subclass
18.
19. // --- all classes are objects
20.   Object o1 = new ExtendingSimpleClass(5);
21.   Object[] o2 = new String[7];
22.
23.   }  // end of ExtendingDemo.main()
24. }    // end of class ExtendingDemo
25.
26. // *** ************************** ***
27. // *** class ExtendingSimpleClass ***
28. // *** ************************** ***
29.
30. class ExtendingSimpleClass extends SimpleClass {
31.
```

```
32. public ExtendingSimpleClass(int topSecret)
33.  {
34.  super(topSecret);             // calls the constructor of the parent class
35.  }
36. public void secretPrinting()
37.  {
38.  System.out.printf("printing from ExtendingSimpleClass: %d\n\n",getTopSecret());
39.  }
40. }   // end of class ExtendingSimpleClass
41.
42. // *** ***************** ***
43. // *** class SimpleClass ***
44. // *** ***************** ***
45.
46. class SimpleClass {
47. private int topSecret;                                          // a private variable
48. public SimpleClass(int topSecret){ this.topSecret = topSecret; } // a constructor
49. public int getTopSecret(){ return topSecret; }                  // a standard getter
50. }   // end of class SimpleClass
51.
```

the main method

- **`main()`** can run without instatiation being *static* and *public*
- the **`main()`** method can also be called explicitly, as any other *static* function

```
 1. public class MainDemo {
 2.
 3. /** A static variable counting the number of times main() was called.
 4.   */
 5. public static int counter = 0;
 6.
 7. /** A non-static variable, can only be accessed when
 8.   * MainDemo has been instatiated.
 9.   */
10. public String myName;
11.
12. /** Constructor for MainDemo,
13.   * this.myName refers to the member variable of the
14.   * current (this) instantion.
```

```
15.   */
16. public MainDemo()
17.   {
18.   this.myName = "I am class number " + MainDemo.counter;
19.   }
20.
21. /** The main() function is called when starting.
22.   * Per definitions it has an String array as argument.
23.   */
24. public static void main(String[] args)
25.   {
26.
27. // note the two ways to access:
28. // MainDemo.counter : always possible
29. //          counter : only from inside MainDemo
30.   MainDemo.counter++;
31.   System.out.printf("number of times main() was called: %d\n",counter);
32.
33. // MainDemo itself can be instatiated
34.   MainDemo newMainDemo = new MainDemo();
35.   System.out.printf("              name of newMainDemo: %s\n",
36.                     newMainDemo.myName);
37.
```

```
38. // the main method is static and can be called
39. // with a string array as an argument,
40. // the if-condition avoids an infinite recursion loop
41.   if (MainDemo.counter < 7)
42.     MainDemo.main(new String[2]);
43.
44.   }  // end of MainDemo.main()
45. }  // end of class MainDemo
```

objects of objects

- Any user defined object, of arbitray complexity, can by used by any other object, eg by arrays

```
 1. public class ObjectOfObjects {
 2.
 3. /** Escape sequences for colored console output,
 4.   * may be operating system dependent.
 5.   * Just for the fun of it.
 6.   */
 7. public static final String ANSI_BLACK   = "\u001B[30m";
 8. public static final String ANSI_RED     = "\u001B[31m";
 9. public static final String ANSI_GREEN   = "\u001B[32m";
10. public static final String ANSI_YELLOW  = "\u001B[33m";
11. public static final String ANSI_BLUE    = "\u001B[34m";
12. public static final String ANSI_MAGENTA = "\u001B[35m";
13. public static final String ANSI_CYAN    = "\u001B[36m";
14. public static final String ANSI_WHITE   = "\u001B[37m";
15. public static final String ANSI_RESET   = "\u001B[m";
16.
```

```
17. public static void main(String[] args)
18.   {
19.
20. // --- create array of ColoredObject
21.   ColoredObject[] manyObjects = new ColoredObject[3];
22.
23. // --- instantiate the individual array elements
24.   manyObjects[0] = new ColoredObject("I am red (hopefully) ",
25.                                      ObjectOfObjects.ANSI_RED);
26.   manyObjects[1] = new ColoredObject("sure, but I am green ",
27.                                      ObjectOfObjects.ANSI_GREEN);
28.   manyObjects[2] = new ColoredObject("yes, but blue is best",
29.                                      ObjectOfObjects.ANSI_BLUE);
30.
31. // --- print colored objects line by line
32. // --- ANSI_RESET resets the output color
33.   System.out.println(" ");
34.   for (int i=0; i<manyObjects.length; i++)
35.     {
36.     String outString = manyObjects[i].getColor()
37.                      + manyObjects[i].getText()
38.                      + ObjectOfObjects.ANSI_RESET;
39.     System.out.println(outString);
```

```
40.     }
41.
42.   System.out.println(" ");
43.   System.out.println("and I am boring black (again)");
44.   System.out.println(" ");
45.
46.   }  // end of ObjectOfObjects.main()
47. }  // end of class ObjectOfObjects
48.
49. // *** ******************** ***
50. // *** class ColoredObject ***
51. // *** ******************** ***
52.
53. class ColoredObject {
54.
55. private String myAnsiColor;
56. private String myText;
57.
58. public ColoredObject(String text, String color)
59.   {
60.   this.myText = text;
61.   this.myAnsiColor = color;
62.   }
```

```
63.
64. public String getColor(){return this.myAnsiColor;}
65. public String getText(){return this.myText;}
66.
67. }  // end of class ColoredObjects
```

reference to objects

- everything, besides primitive data types
  int, **double**, **boolean**, ... , is an object
- the name of an object is a reference to it,
  the place in memory where the object starts,
  and **not** the object itself
- **String** objects are special:
  every single **String** operation creates a new **String** object

```
1. public class ReferenceDemo {
2.
3. public static void main(String[] args)
4.   {
5.
6. // aInt and bInt are two distinct integer variables
7.   int aInt = 1;
8.   int bInt = aInt;
9.   System.out.printf("aInt, bInt:  %d | %d\n",aInt,bInt);
```

```
10.   bInt = 2;
11.   System.out.printf("aInt, bInt:  %d | %d\n",aInt,bInt);
12.   System.out.println(" ");
13.
14. // bArray is a synonym for aArray
15. // bArray is not a new object
16.   int[] aArray = { 1, 2, 3};
17.   int[] bArray = aArray;
18.   System.out.printf("aArray: %d %d %d\n",aArray[0],aArray[1],aArray[2]);
19.   System.out.printf("bArray: %d %d %d\n",bArray[0],bArray[1],bArray[2]);
20.   System.out.println(" ");
21.
22.   aArray[0] = 0;
23.   System.out.printf("aArray: %d %d %d\n",aArray[0],aArray[1],aArray[2]);
24.   System.out.printf("bArray: %d %d %d\n",bArray[0],bArray[1],bArray[2]);
25.   System.out.println(" ");
26.
27.   bArray[2] = 7;
28.   System.out.printf("aArray: %d %d %d\n",aArray[0],aArray[1],aArray[2]);
29.   System.out.printf("bArray: %d %d %d\n",bArray[0],bArray[1],bArray[2]);
30.   System.out.println(" ");
31.
32. // String operations always create new String objects
```

```
33.   String aStr = "a comes first";
34.   String bStr = aStr;           // a new String object is created implicitly
35.   System.out.printf("aStr, bStr:  %s | %s\n",aStr,bStr);
36.
37. // creating a new String object 'b comes second' and
38. // assigning its reference to the variable bStr
39.   bStr = "b comes second";
40.   System.out.printf("aStr, bStr:  %s | %s\n",aStr,bStr);
41.   System.out.println(" ");
42.
43.   }  // end of ReferenceDemo.main()
44. }  // end of class ReferenceDemo
```

call by value / call by reference

call by value

When a function **f(*arg*)** is called only the value of the argument *arg* is passed, not its references.
Changes of the argument inside the function **f()** **do not change** the value of the argument outside of the function.

call by reference

When a function **f(*arg*)** is called only the reference of *arg* is passed to the function.
Changes of the argument inside the function **f()** **do change** the value of the argument outside of the function.

JAVA

- primitive data types -- call by value
- objects -- call by reference

```
1. public class FunctionCallDemo {
2.
```

```
 3. /** Changing the value of a primitive data type has
 4.   * has no effect outside the residing function,
 5.   * call-by-value only passes the value and not the
 6.   * reference of a primitive data type.
 7.   */
 8. public static void callByValue(int inputInteger)
 9.   {
10.   System.out.printf("in FunctionCallDemo.callByValue(), "
11.                     + "inputInteger : %d\n",inputInteger);
12.   inputInteger = 0;
13.   }
14.
15. /** Only the reference, not the value, of Objects are
16.   * passed to the function, creating a synonym of the argument.
17.   */
18. public static void callByReference(int[] inputArray)
19.   {
20.   inputArray[0] = 0;
21.   }
22.
23. public static void main(String[] args)
24.   {
25.
```

```
26. // --- call by value, for primitive data types
27.   int callInteger = 10;
28.   System.out.println("  ");
29.   FunctionCallDemo.callByValue(callInteger);
30.   System.out.printf("       in FunctionCallDemo.main(), "
31.                   + " callInteger : %d\n",callInteger);
32.   System.out.println("  ");
33.
34. // --- call by value, for objects
35.   int[] callArray = { 33, 44, 55};
36.   System.out.printf("       in FunctionCallDemo.main(), "
37.                   + "   callArray : %2d %2d %2d\n",
38.                   callArray[0], callArray[1], callArray[2]);
39.   FunctionCallDemo.callByReference(callArray);
40.   System.out.printf("       in FunctionCallDemo.main(), "
41.                   + "   callArray : %2d %2d %2d\n",
42.                   callArray[0], callArray[1], callArray[2]);
43.
44.   }  // end of FunctionCallDemo.main()
45. }  // end of class FunctionCallDemo
```

classes and files

one file per class

- standard: file *MyClass* only contains the **class** *MyClass*
- idea: classes can be used by many programs, interchangeability
- needed: one directory per project

one program per file

- best for small/medium projects:
  file *MyProgram* contains the **class** *MyProgram* and all (self written) referenced classes
- advantage: reuse (after an extended period of inactivity) and transfer (eg via email) straightforward

```
1. public class MyProgram {
2.
3. public static void main(String[] args)
```

```
 4.   {
 5.   MyClass mc = new MyClass();
 6. // ...
 7.   }
 8. }
 9.
10. // *** ************* ***
11. // *** class MyClass ***
12. // *** ************* ***
13.
14. class MyClass {
15. // ...
16. }
```

interfaces

- **API** - application programming interface
  a common interface (standard) for possibly many applications and/or realizations
- defines abstract methods without implementing them
- a **`class`** implenting an interface must (mandatory)
  overwrite all methods of an **`interface`**
- a **`class`** can can implement many interfaces

```
1. interface MovingHero {
2.
3. public String getSymbol();       // symbol of hero
4. public int getPosition();        // position of hero
5. public void move();              // move the hero
6. public void invertDirection();   // invert direction of movement
7.
8. }
```

interface demo - inline battle

- Unicode symbols like *\u0040* can be printed
- the control sequence *\r* moves the pointer (for writing to the output) to the start of the line
- exercise: add a green hero

```
 1. public class InterfaceDemo {
 2.
 3. /** The length of the battlefield.
 4.   */
 5. public static final int length = 30;
 6.
 7. public static void main(String[] args)
 8.                     throws Exception
 9.   {
10.
11. // -- create array of hero (interfaces, not classes
12.   MovingHero[] allHeros = new MovingHero[2];
```

```
13.
14. // -- instantiate the heros, with classe now
15.   allHeros[0] = new BlueHero(InterfaceDemo.length);
16.   allHeros[1] = new RedHero(InterfaceDemo.length);
17.
18. // -- the configuration of the heros on the battlefield
19.   String[] heroPos = new String[InterfaceDemo.length];
20.
21.   System.out.println(" ");
22.
23. // -- let the heros battle each other
24.   for (int i=0; i<100; i++)
25.     {
26.     for (int h=0; h<heroPos.length; h++)            // empty battlefield
27.       heroPos[h] = " ";
28.     for (int a=0; a<allHeros.length; a++)           // loop over all heros
29.       {
30.       int pos = allHeros[a].getPosition();
31.       if (heroPos[pos].equals(" "))
32.         heroPos[pos] = allHeros[a].getSymbol();  // draw hero
33.       else
34.         allHeros[a].invertDirection();           // invert if colliding
35.       allHeros[a]. move();                       // move them
```

```
36.       }
37.
38.     for (int l=0; l<heroPos.length; l++)            // print battlefield
39.       System.out.printf("%s",heroPos[l]);
40.
41. // --- no linebreak, return to beginning of line
42.     System.out.print("\r");
43.
44. // --- pause the program execution (ms)
45.     Thread.sleep(200);
46. //
47.     } // end of loop over all steps
48.
49.
50.   } // end of InterfaceDemo.main()
51. } // the dof class InterfaceDemo
52.
53. // *** ********************** ***
54. // *** Interface moving heros ***
55. // *** ********************** ***
56.
57. /** Any here hero is a moving hero.
58.   */
```

```
59. interface MovingHero {
60.
61. public String getSymbol();       // symbol of hero
62. public int getPosition();        // position of hero
63. public void move();              // move the hero
64. public void invertDirection();   // invert direction of movement
65.
66. }  // end of interface MovingHero
67.
68. // *** ************** ***
69. // *** class red hero ***
70. // *** ************** ***
71.
72. class RedHero implements MovingHero {
73.
74. private static final String mySymbol = "\u001B[31m" + "\u0040" + "\u001B[m";
75. private int myPos;
76. private int mySpeed = 2;
77. private int length;
78.
79. public RedHero (int length)
80.   {
81.   this.myPos = (int)(Math.random()*length);
```

```
 82.   this.length = length;
 83.   }
 84.
 85. public String getSymbol()
 86.   {
 87.   return this.mySymbol;
 88.   }
 89. public int getPosition()
 90.   {
 91.   return this.myPos;
 92.   }
 93. public void move()
 94.   {
 95.   this.myPos = (this.myPos+this.mySpeed+this.length)%this.length;
 96.   }
 97. public void invertDirection()
 98.   {
 99.   this.mySpeed *= -1;
100.   }
101.
102. }  // end of class RedHero
103.
104. // *** *************** ***
```

```
105. // *** class blue hero ***
106. // *** *************** ***
107.
108. class BlueHero implements MovingHero {
109.
110. private static final String mySymbol = "\u001B[34m" + "\u0026" + "\u001B[m";
111. private int myPos;
112. private int length;
113.
114. public BlueHero (int length)
115.   {
116.   this.myPos = (int)(Math.random()*length);
117.   this.length = length;
118.   }
119.
120. public String getSymbol()
121.   {
122.   return this.mySymbol;
123.   }
124. public int getPosition()
125.   {
126.   return this.myPos;
127.   }
```

```
128. public void move()
129.   {
130.   this.myPos = (this.myPos + 1)%this.length;
131.   }
132. public void invertDirection()
133.   {
134.   }
135.
136. }  // end of class BlueHero
```