Chapter 10

Code Efficiency

What is in This Chapter ?

In this set of notes, we concentrate on aspects of programming that can make our code more efficient. We begin by explaining how to **exit our loops** when we have found what we needed from them. Then we discuss how to simplify our **IF** statements through a proper understanding of **booleans**. We further discuss **enumerated types** and when they should be used. Further, we discuss the notion of objects **equality vs. identity** and how understanding the differences can allow us to efficiently share objects without problems, thereby reducing store space and run time. This topic brings up the need to write an **equals()** method to compare two objects. Finally, we discuss **arrays** and how they can be used to write code that is a little more efficient than **ArrayLists**.

10.1 Exiting Loops Efficiently

Consider a piece of code that loops through an **ArrayList** of **Strings** to find a specific item. We may need to do this, for example, when we look up someone's name in a list to see whether or not they are registered. Here is a simple method that does such a search:

```
boolean isRegistered(String searchName) {
    for (String name: registeredList) {
        if (name.equals(searchName))
            return true;
    }
    return false;
}
```

Notice that the method returns **true** the moment it finds the name that it is searching for. In the case where the name is not in the list, then the FOR loop must iterate through the entire list, returning **false** afterwards. This code is efficient, because it stops searching as soon as it finds what it is looking for. However, consider now a method that contains this search loop, but then needs to do something afterwards:

```
void doSomething(String searchName) {
    boolean found = false;

    for (String name: registeredList) {
        if (name.equals(searchName))
            found = true;
    }
    if (found)
        // do something ...
    else
        // do something else ...
}
```

This looping code is no longer time-efficient. Do you know why ? Imagine that the **registeredList** has 100,000 names in it. Assume that the first name is the one that matches the **searchName**. The FOR loop will still continue checking the remaining 99,999 names, even though it has already found what it is looking for!

There is an efficient way to exit a loop when we know that we no longer need to keep checking. The **break** keyword in JAVA will "break out of" (i.e., exit) the loop that it is inside of. Hence, we can insert into the above code a **break** statement as follows …

```
void doSomething(String searchName) {
    boolean found = false;

    for (String name: registeredList) {
        if (name.equals(searchName)) {
            found = true;
            break;
        }
    }
    if (found)
        // do something ...
    else
        // do something else ...
}
```

Jumps out of loop.

Another situation that we may want to exit from a loop is when a certain condition occurs. For example, we may want to loop until the user enters a valid number:

```
int         num = 0;
boolean     valid = false;

while(!valid) {
    System.out.println("Enter the percentage: ");
    num = new Scanner(System.in).nextInt();
    if ((num >= 0) && (num <= 100))
        valid = true;
    else
        System.out.println("Invalid Entry");
}
```

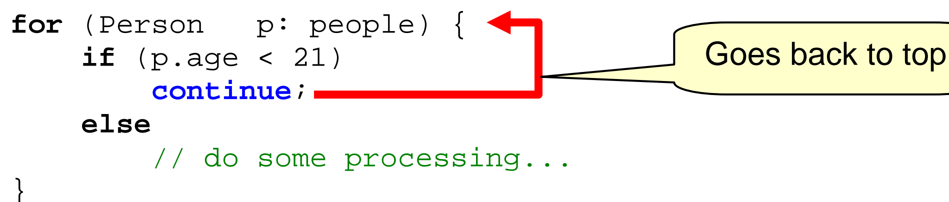
In the code above, we use the **valid** boolean variable to indicate whether or not the number entered by the user was in the valid range of 0 to 100. The while loop simply repeats until this **boolean** (known as a “flag”) changes its state from **false** to **true**.

We can actually eliminate the need for the **boolean** variable by making use of the **break** statement as follows:

```
int   num = 0;

while(true) {
    System.out.println("Enter the percentage: ");
    num = new Scanner(System.in).nextInt();
    if ((num >= 0) && (num <= 100))
        break;
    System.out.println("Invalid Entry");
}
```

Notice that the code above uses a potentially infinite **while** loop. However, the loop will exit when the **break** statement is reached, provided that the user enters a valid number. This is equally efficient in terms of the number passes through the loop, but certainly the use of the **break** statement reduces and simplifies the code.

In addition to the **break** statement (which exits a loop), JAVA offers a **continue** statement which does not exit the loop, but instead goes back to the top of the loop to do another iteration. This is useful if we need to do something like ignore certain entries in a list. For example, assume that we wanted to go through a list of people and process information for all people in the list that are 21 years of age or older:

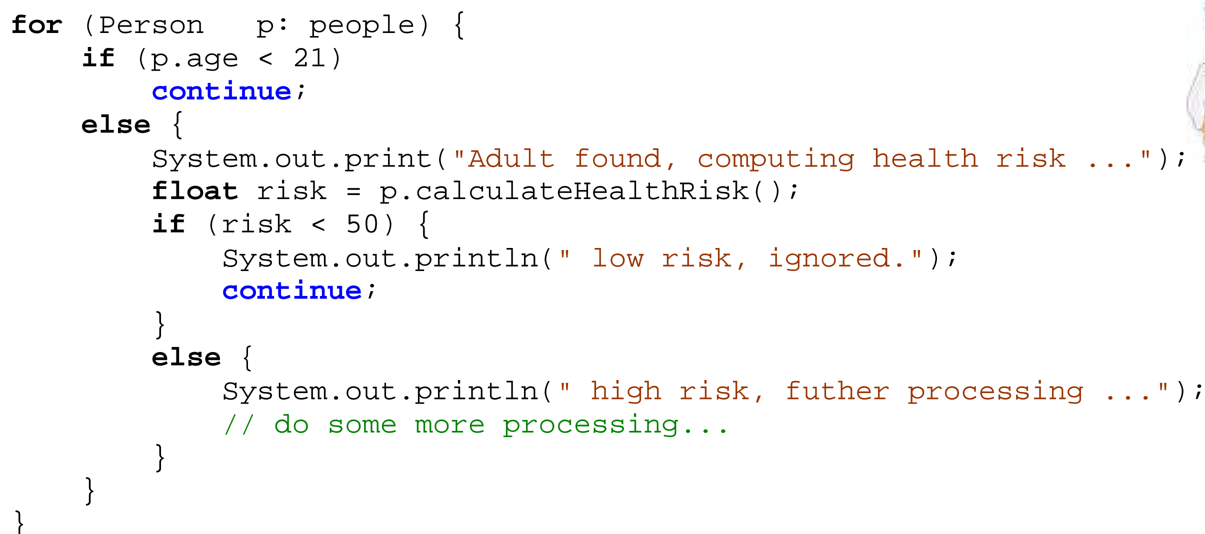



```
for (Person   p: people) {
    if (p.age < 21)
        continue;
    else
        // do some processing...
}
```

In the above code, the **continue** statement goes back and gets the next person, evaluating the loop again. Of course, we could have done this without the **continue** as follows:

```
for (Person   p: people) {
    if (p.age >= 21) {
        // do some processing...
    }
}
```

However, sometimes the code is more complicated. For example, assume that we want to ignore all people under 21 or whose health risk factor is below 50%:



```
for (Person   p: people) {
    if (p.age < 21)
        continue;
    else {
        System.out.print("Adult found, computing health risk ...");
        float risk = p.calculateHealthRisk();
        if (risk < 50) {
            System.out.println(" low risk, ignored.");
            continue;
        }
        else {
            System.out.println(" high risk, futher processing ...");
            // do some more processing...
        }
    }
}
```

Notice above that the **continue** statement may be used as many times as necessary to abandon this particular "round" through the loop and move onto the next person. This saves processing time, which may be significant if the list of people is large.

We can also add ***labels*** in JAVA to indicate which loop we want to get out of in the case of multiple/nested loops. Consider searching a table with rows and columns of potentially large amounts of data at each (row/column) cell. We may want to abandon the processing of a cell in the table if we find that further processing of the cell, row, or column is not needed:

```
rows:
for (int row=0; row<1000; row++) {

    columns:
    for (int col=0; col<1000; col++) {
        ...
        if (someConditionOccursThatMakesThisColumnUseless())
            continue columns;

        if (someConditionOccursThatMakesThisRowUseless())
            continue rows;      // same as break columns;

        if (someConditionOccursThatRequiresStoppingAltogether())
            break rows;
        ...
    }
}
```

10.2 Proper Use of Booleans

Another aspect of proper coding style pertains to simplifying your code by making proper use of **booleans**. Although the **IF** statement is quite easy to use, it is often the case that some students do not **fully** understand its logic when **booleans** are being used. As a result, many students end up writing overly complex and inefficient code. Also, the **IF** statement is often used when it is not even required !!

To illustrate this, consider the following examples of "BAD" coding style. Try to determine why the code is inefficient and how to improve it. If it is your desire to be a good programmer, pay careful attention to these examples.

Example 1:

```
boolean male = ...;

if (male == true) {
    System.out.println("male");
else
    System.out.println("female");
```

Here, the **boolean** value of **male** is *already* **true** or **false**, we can make use of this fact:

```
boolean male = ...;

if (male) {
    System.out.println("male");
else
    System.out.println("female");
```

Example 2:

```
boolean adult = ...;

if (adult == false)
    discount = 3.00;
```

Here is a similar situation as above, but with a negated **boolean**. Below is better code.

```
boolean adult = ...;

if (!adult) {
    discount = 3.00;
```

Example 3:

```
boolean tired = ...;

if (tired)
    result = true;
else
    result = false;
```

Above, we are actually returning the identical **boolean** as **tired**. No **if** statement is needed:

```
boolean tired = ...;

result = tired;
```

Example 4:

```
boolean discount;

if ((age < 6) || (age > 65))
    discount = true;
else
    discount = false;
```

The discount is solely determined by the **age**. No **if** statement is needed:

```
boolean discount;

discount = (age < 6) || (age > 65);
```

Example5:

```
boolean fullPrice;

if ((age < 6) || (age > 65))
    fullPrice = false;
else
    fullPrice = true;
```

Just like above, we do not need the **if** statement:

```
boolean fullPrice;

fullPrice = !((age < 6) || (age > 65));
```

or …

```
boolean fullPrice;

fullPrice = (age >= 6) && (age <= 65);
```

10.3 Enumerated Types

Consider writing a program that evaluates certain code depending on the day of the week:

```
String    dayOfWeek = ...;

if (dayOfWeek.equals("SATURDAY") || dayOfWeek.equals("SUNDAY"))
    // do something ...
else
    // do something else ...
```

Each time we make a **String** comparison using the **equals()** method, this takes time because JAVA needs to compare the characters from the strings. However, comparing integers is fast. So we could define some integer constants for the days of the week, store our day of the week as an integer and then and simply compare the constants using **==** as follows:

```
static final int MONDAY = 1;
static final int TUESDAY = 2;
static final int WEDNESDAY = 3;
static final int THURSDAY = 4;
static final int FRIDAY = 5;
static final int SATURDAY = 6;
static final int SUNDAY = 7;

...
...
int  dayOfWeek = ...;

if ((dayOfWeek == SATURDAY) || (dayOfWeek == SUNDAY))
    // do something ...
else
    // do something else ...
```

This comparison is very fast since it merely compares two integers as opposed to a sequence of characters. We should use this strategy whenever we have a set of fixed constant values such as the days of the week, planets in the solar system, choices on a menu, command-line flags, etc..

Rather than clutter up our code with these constants, we can define them in their own publicly accessible class as follows …

```
public class Day {
    public static final int MONDAY = 1;
    public static final int TUESDAY = 2;
    public static final int WEDNESDAY = 3;
    public static final int THURSDAY = 4;
    public static final int FRIDAY = 5;
    public static final int SATURDAY = 6;
    public static final int SUNDAY = 7;
}
```

Then we can make use of this class in our code:

```
...
int  dayOfWeek = ...;

if ((dayOfWeek == Day.SATURDAY) || (dayOfWeek == Day.SUNDAY))
    // do something ...
else
    // do something else ...
```

JAVA provides a useful keyword called **enum** that can be used for this exact situation. The **enum** keyword can be used in place of the **class** keyword to define a set of constant symbols such as the days of the week. It makes the code much simpler:

```
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

I'm sure you will agree that this is much shorter than the class definition. In fact, we can define as many of these enumerated types as we would like … each in their own file:

```
public enum Gender {
    MALE, FEMALE
}
```

```
public enum Size {
    TINY, SMALL, MEDIUM, LARGE, HUGE
}
```

```
public enum Direction {
    NORTH, NORTH_EAST, EAST, SOUTH_EAST, SOUTH,
    SOUTH_WEST, WEST, NORTH_WEST
}
```

These would be saved, compiled and then used in our programs just as we would use any other class or interface definition. Once the type is defined, we can even define variables with these types, assigning them one of the values in the list specified in the **enum** declaration:

```
Gender     gender;
Size       orderSize;
Weekday    payday;
Direction  homePosition;

gender = Gender.MALE;
orderSize = Size.MEDIUM;
payday = Weekday.FRIDAY;
homePosition = Direction.NORTH_WEST;
```

Notice that the variable type matches the **enum** type. Also notice that when assigning a value to the variable, we must again specify the **enum** type followed by the dot **.** operator and then by one of the fixed constant values.

If we wanted to, we could have defined the **enum** types right inside of our class (but then they may not be declared **public** and would only be useable within that class's code). It would however, allow us to reduce the number of classes that we write. To use the **enum** types in your own class definition, you should place them directly under the class definition (or above it). You are not allowed to place the declarations in the executable (i.e., running) part of your program (i.e., in the main method, nor in other methods). Note that when you display an **enum** type value (as in the code above), the symbol value is printed out just as it appears in the type declaration.

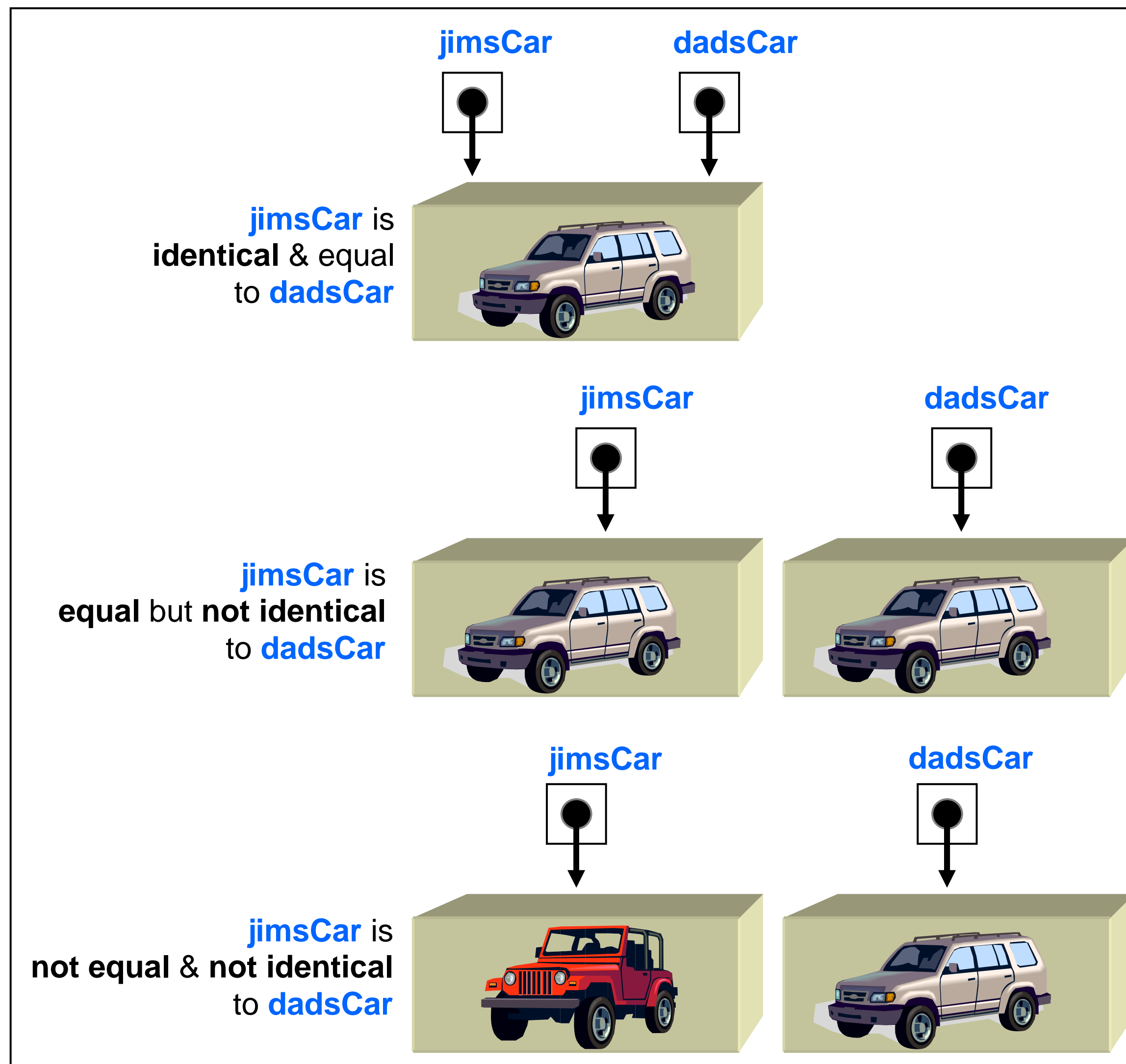
```
public class EnumTestProgram {

    enum Gender {MALE, FEMALE};
    enum Size {TINY, SMALL, MEDIUM, LARGE, HUGE};
    enum Weekday {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
                  SATURDAY, SUNDAY};
    enum Direction {NORTH, NORTH_EAST, EAST, SOUTH_EAST,
                    SOUTH, SOUTH_WEST, WEST, NORTH_WEST};

    public static void main(String args[]) {
        Gender     gender = Gender.MALE;
        Size       orderSize = Size.MEDIUM;
        Weekday    payday = Weekday.FRIDAY;
        Direction  homePosition = Direction.NORTH_WEST;

        System.out.println(gender);         // displays MALE
        System.out.println(orderSize);      // displays MEDIUM
        System.out.println(payday);         // displays FRIDAY
        System.out.println(homePosition);   // displays NORTH_WEST
    }
}
```

10.4 Equality Vs. Identity

From childhood, all of us have been taught to *share* with one another. Not only does this instill proper moral values in us, but it also saves on costs (e.g., one toy can be shared by many children, instead of buying them each their own). Sharing is even a good idea when programming. Since created objects use up memory space, it is often desirable to share objects (or information) whenever possible.

Since objects can be shared, it is important to know whether or not the same object is being used in two or more places. This brings up the notion of ***equality*** vs. ***identity***. In JAVA, two objects are considered to be ***equal*** if they are of the "same type" and have the "same attribute values". That is, if the values of the instance variables of the two objects are the same, then the objects are equal. Two objects are ***identical*** if and only if they are the exact same object.

For example, here is how equality and identity differ in the situations where Jim and his dad share a family car, buy two cars of the same kind or buy two different kinds of cars:



So the word **identical** means the "exact same object" in JAVA (i.e., the two objects point to the same memory location). Note that this is different from the English language definition of identity.

Equality actually is very subjective. For example, if you lose a 5 year old child's teddy bear and then buy her/him a new one of the same type, the child will likely agree that they are not equal. In JAVA, we will spend a lot of time defining our own objects, so **we** are the ones that will decide what it actually means for two of our objects to be equal.

Consider this code which creates three people:

```
Person     p1, p2, p3;

p1 = new Person("Hank", "Urchif", 19, 'M', false);
p2 = new Person("Hank", "Urchif", 19, 'M', false);
p3 = new Person("Holly", "Day", 67, 'F', true);
```

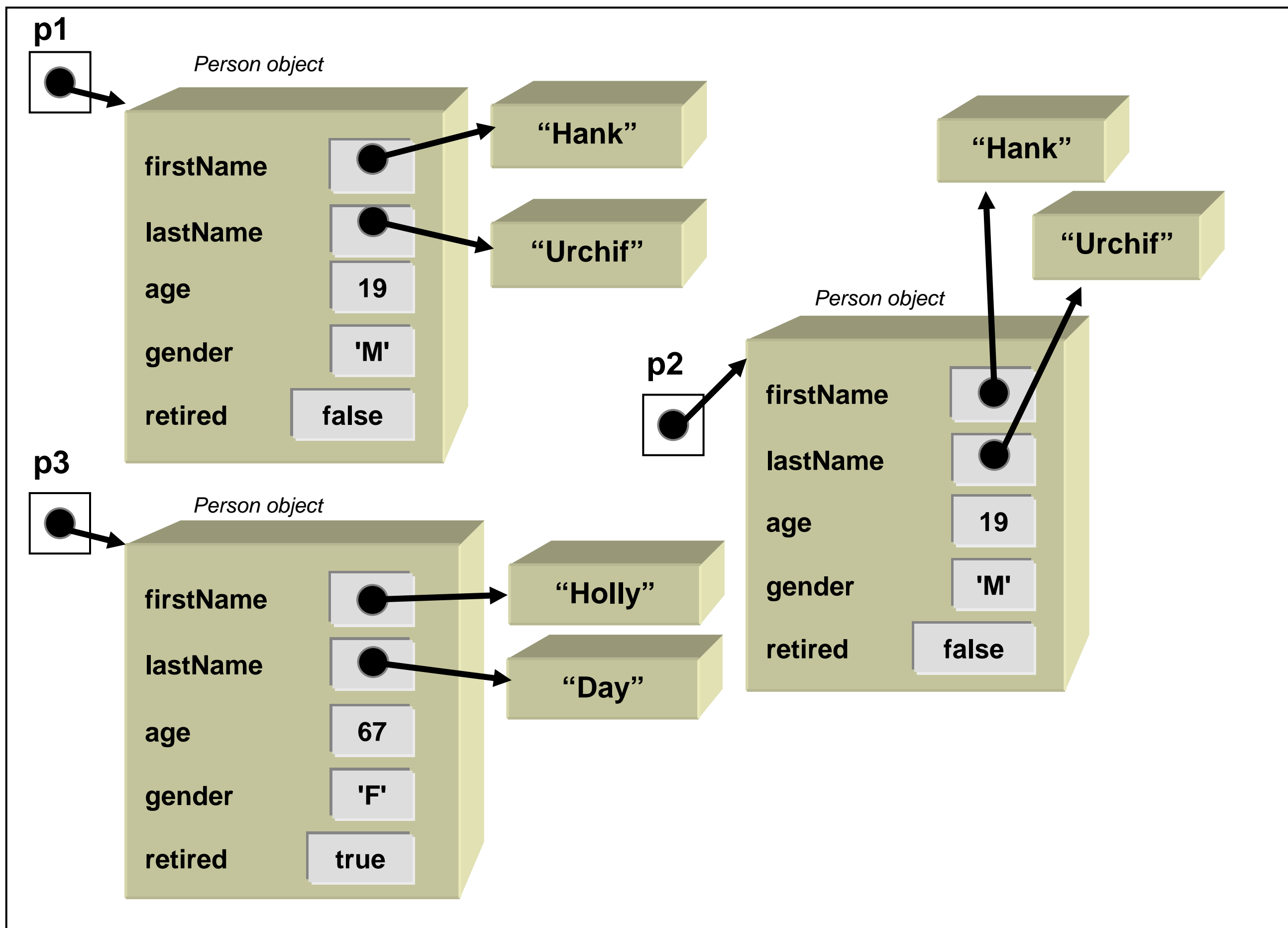
If we want to ask whether or not two of these people are *equal*, intuitively we would use the **==** operator as follows:

```
if (p1 == p2)
    System.out.println("p1 and p2 are the same two people");
else
    System.out.println("p1 and p2 are different people");

if (p1 == p3)
    System.out.println("p1 and p3 are the same two people");
else
    System.out.println("p1 and p3 are different people");
```

But if we evaluate the following code, we would get the following result:

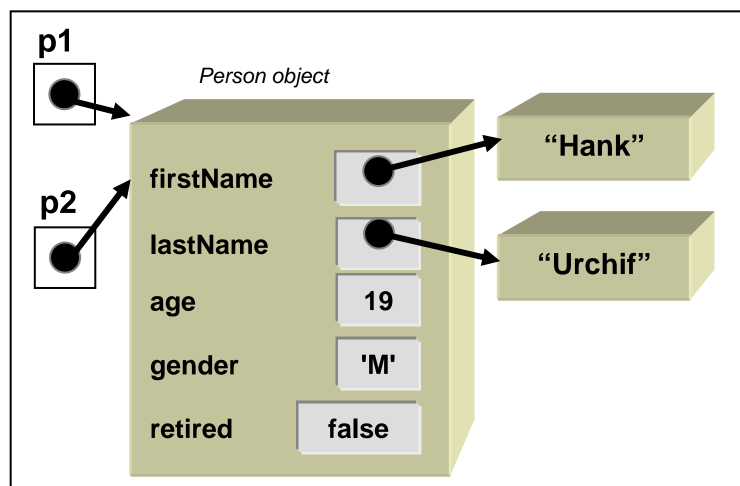
```
p1 and p2 are different people
p1 and p3 are different people
```

Why is **p1** not equal to **p2** ? Clearly both people are **equal** since they have the same **firstName**, **lastName**, **age**, **gender** and **retirement status**. However, in JAVA, when the **==** operator is used with objects, it means "identity" ... not "equality". That is, the **==** operator determines whether or not the two variables are pointing to the exact same object. Here is a diagram showing how the variables are related to the objects …



Notice that **p1** and **p2** are pointing to different objects, although the objects have the same attribute values. As it turns out, whenever we make a new instance of an object, we actually get a unique object every time, thus that new object can only be ***identical*** to itself.

So what kind of coding situation would allow **(p1 == p2)** to return **true** ? Well, we would have to have code that allows the two variables to refer to the same object as follows:



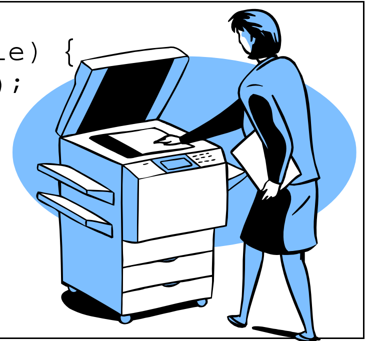
We can do this with the following code:

```
Person     p1, p2;

p1 = new Person("Hank", "Urchif", 19, 'M', false);
p2 = p1;
```

Now **p1** is identical to **p2** and so **(p1 == p2)** will return **true**.

The above code is just an example to illustrate how to make two variables point to the same object. A more realistic example may be to copy over the contents of an **ArrayList**. For example, consider the following method which creates and returns a copy of an **ArrayList** containing **Person** objects:

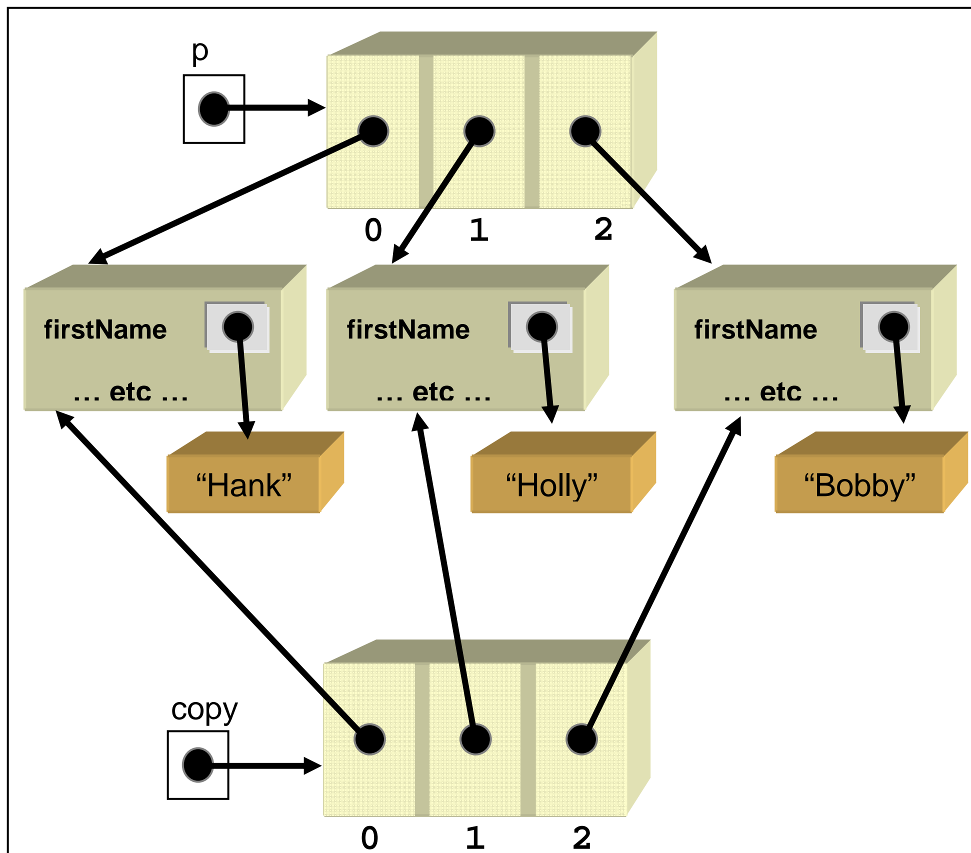
```
public ArrayList<Person> makeCopy(ArrayList<Person> people) {
    ArrayList<Person>   newList = new ArrayList<Person>();

    for(Person  p: people)
       newList.add(p);

    return newList;
}
```



Consider testing this method as follows:

```
ArrayList<Person>   p, copy;

p = new ArrayList<Person>();
p.add(new Person("Hank", "Urchif", 19, 'M', false));
p.add(new Person("Holly", "Day", 67, 'F', true));
p.add(new Person("Bobby", "Socks", 8, 'M', false));

copy = makeCopy(p);
```

Here is what we accomplish with this code …

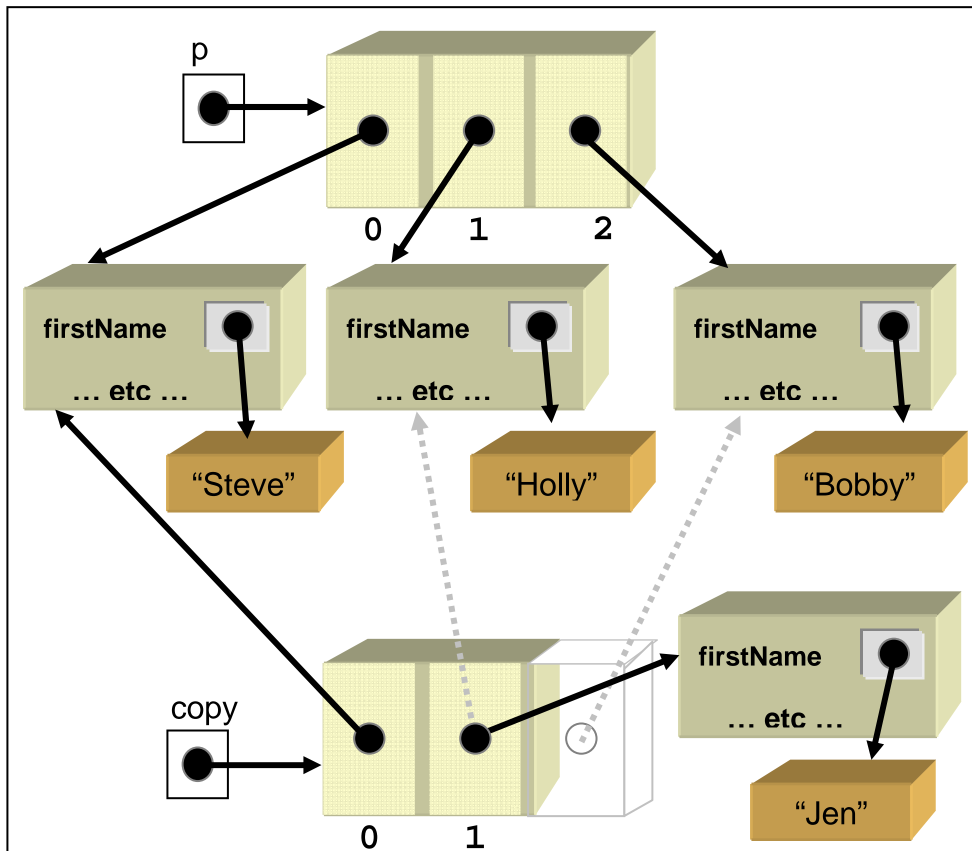


Notice that when we add the **Person** objects to the copy, they are actually shared between the two **ArrayLists**. This is known as a ***shallow copy***. Normally this is not a problem when writing code as long as we know that the items are shared. Consider what happens here:

```
Person               last;
ArrayList<Person>    p, copy;

p = new ArrayList<Person>();
p.add(new Person("Hank", "Urchif", 19, 'M', false));
p.add(new Person("Holly", "Day", 67, 'F', true));
p.add(last = new Person("Bobby", "Socks", 8, 'M', false));

copy = makeCopy(p);

copy.set(1, new Person("Jen", "Tull", 42, 'F', false));
copy.get(0).setFirstName("Steve");
copy.remove(last);
```

Notice the use of the **set()** method for **ArrayLists**. This method will replace the object in the list at the given index (specified by the first parameter .. .in this case **1**) with the new object (given as the second parameter … in this case **Jen**). When *replacing* **Holly** with **Jen** in the **copy** list, the original list remains unchanged. However, when modifying **Hank** from the copy, then the original list is affected since the object that they were sharing has now been altered. Finally, when we remove **Bobby** from the copy, the original remains unchanged.

Here is the diagram showing what happened:

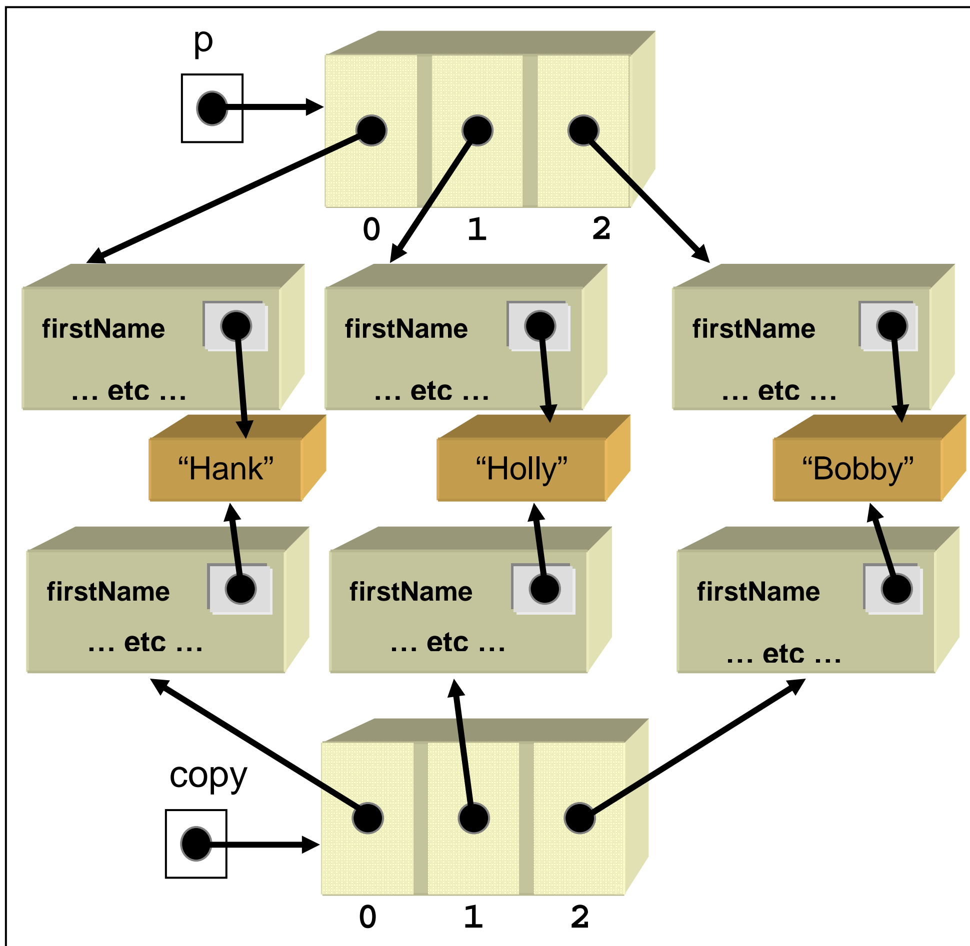


So you can see, by replacing, adding or removing items to a copied list, the original list remains intact. However, when we access a shared object from the copy and go into it to make changes to its attributes, then the original list will be affected.

To make a more separated copy, we could make what is known as a ***deep copy*** of the **ArrayList** so that the **Person** objects are not shared between them (i.e., each list had their own copies of the **Person** objects), then we would have to create new **Person** objects with the same attributes as follows:

```
public ArrayList<Person> makeCopy(ArrayList<Person> people) {
    ArrayList<Person>   newList = new ArrayList<Person>();

    for(Person  p: people) {
        Person  x = new Person();
        x.firstName = p.firstName;
        x.lastName = p.lastName;
        x.age = p.age;
        x.gender = p.gender;
        x.retired = p.retired;
        newList.add(x);

    return newList;
}
```

Here is what the **ArrayLists** would look like if we did **copy = makeCopy(p);**



Notice that the Strings are still shared! To make a truly separated copy of the list, we would need to copy the strings as well like this:

```
x.firstName = new String(p.firstName);
x.lastName = new String(p.lastName);
```

In fact, in order to make a truly deep copy, we would need to ensure that all objects of all attributes are themselves copied.

You should now understand what it means for two objects to be identical and you should understand how to make two or more variables point to identical objects as well as how to make sure that they point to unique objects by copying them.

Recall, that we use **==** to determine whether or not two objects are identical:

```
Person     p1 = new Person("Hank", "Urchif", 19, 'M', false);
Person     p2 = new Person("Hank", "Urchif", 19, 'M', false);

if (p1 == p2)
    System.out.println("p1 and p2 are the same two people");
else
    System.out.println("p1 and p2 are different people");
```

If we want to ask whether or not these two people are **equal**, instead of identical, we replace the **==** operator with a call to **equals()** as follows:

```
Person     p1, p2;

p1 = new Person("Hank", "Urchif", 19, 'M', false);
p2 = new Person("Hank", "Urchif", 19, 'M', false);

if (p1.equals(p2))
    System.out.println("p1 and p2 are equal");
else
    System.out.println("p1 and p2 are not equal");
```

As it turns out, all objects understand the **equals(Object x)** method in JAVA. That is, there is an **equals()** method defined in the **Object** class that all objects will inherit. However, in the above code, the call to **equals()** still returns **false**, even though all of the attributes are the same for both **Person** objects!! Why ? Well, here is what the code does inside the **equals()** method that all objects inherit:

```
public boolean equals(Object x) {
   return this == x;
}
```

So, when we call **equals()**,JAVA simply returns **true** if the objects are identical and **false** otherwise. So, the default **equals()** method checks whether two objects are *identical*, not just *equal*. But this does not help us at all since it is no different from simply using the **==** operator.

As a standard programming convention, when we define our own objects, we should also define our own **equals()** method for that object which overrides the default **equals()** method from the **Object** class. Many existing JAVA classes already have a different implementation of the **equals()** method that properly checks for equality.

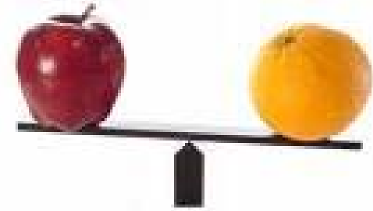
Let us look at how to write an **equals()** method for our **Person** object. First of all, the **equals()** method MUST be written in the **Person** class, it must be **public**, it must return a **boolean** and it must take a single **Object** parameter as follows:

```
public class Person {
    ...
    public boolean equals(Object x) {
        ...
    }
    ...
}
```

If we make a mistake in the method's signature (e.g., we make a mistake in the return type, name or parameter), then the method will not be called by JAVA and it will use the old (i.e., default) one instead of ours.

So, now what do we write in the method ? Well, it must return **true** if the objects are equal and **false** otherwise. Notice that the parameter is of type **Object**. That means, we can pass in ANY type of object whatsoever. So, we can actually ask if **Person** objects are equal to other **Person** objects or **Car** objects or **Apple** objects or **String** objects etc...

```
Person      p1;
Apple       a;
Car         c;

p1 = new Person("Hank", "Urchif", 19, 'M', false);
a = new Apple("Red");
c = new Car("Red", 1991, "Porsche", "959");

if (p1.equals(a)) {...}           // false
if (p1.equals(c)) {...}           // false
if (p1.equals("Hank")) {...}      // false
```



Of course, it should always return **false** if the parameter object is not a **Person** object. We would need to first check to make sure that the object passed in is a **Person**. We can do this by using the **instanceof** keyword in JAVA:

```
public boolean equals(Object x) {
    if (!(x instanceof Person))
        return false;
    ...
}
```

Notice that the '**o**' in "**of**" is not capitalized. The **instanceof** keyword actually checks whether or not the object on the left (i.e., the **x** parameter in this case) is the same type as a particular class which is specified on the right (i.e., **Person** in this case). As it turns out, objects that belong to a *subclass* of **Person** will also work here, so you should be aware of that if you feel that special subclasses of person should not be considered equal for some reason.

The next step is to ***type-cast*** the parameter object into a **Person** object. This will tell the JAVA compiler that we would like to treat the incoming object as a **Person** object. That is, we want to call methods for it from the **Person** class:

```
public boolean equals(Object x) {
    if (!(x instanceof Person))
        return false;

    Person  p = (Person)x;
    ...
}
```

At this point, we have ensured that the incoming object is indeed a **Person** object and we now have a variable (i.e., **p**) that represents this **Person**. We now just need to decide what we think it means for two Person objects to be equal. To do this, we need to look at the attributes of our **Person** object. Our **Person** objects all have an **firstName**, **lastName**, **age**, **gender** and **retired** status. The equality of two people depends on the application.

For example, if we are searching for a particular person in a list, it may be sufficient just to confirm that the first and last names are equal to the one we are looking for. However, in some cases, two people may have the same name but different ages and retired status (e.g., John Doe, Jane Doe). In that case we should check for more information. As a rule of thumb, it may be best to simply check all attributes for equality.

So in our method, we check each attribute and only return **true** if all attributes are equal:

```
public boolean equals(Object x) {
    if (!(x instanceof Person))
        return false;

    Person  p = (Person)x;
    if ((this.firstName.equals(p.firstName)) &&
        (this.lastName.equals(p.lastName)) &&
        (this.age == p.age) &&
        (this.gender == p.gender) &&
        (this.retired == p.retired))
       return true;
    else
       return false;
}
```

Notice that we check each attribute from the *receiver* **Person** (i.e., **this**, which represents the **Person** for which we call the **equals()** method) to make sure that it is equal to the attribute from the parameter **Person** (i.e., **p**). We use **&&** to ensure that ALL attributes are equal before we return **true**. Notice that the **age**, **gender** and **retired** attributes are all checked using the **==** operator. This is because they are primitives, not objects. The **firstName** and **lastName** attributes are compared using the **equals()** method because they are **String** objects and **==** would only check identity … which is not what we want. Of course, we want to reduce and simplify the code above by noticing the poor use of **booleans** in the **if** statement:

```
public boolean equals(Object x) {
    if (!(x instanceof Person))
        return false;

    Person  p = (Person)x;
    return ((this.firstName.equals(p.firstName)) &&
            (this.lastName.equals(p.lastName)) &&
            (this.age == p.age) &&
            (this.gender == p.gender) &&
            (this.retired == p.retired));
}
```

Assuming that the above method is implemented within the **Person** class, then the following code will now return `"p1 and p2 are equal"`:

```
Person     p1, p2;

p1 = new Person("Hank", "Urchif", 19, 'M', false);
p2 = new Person("Hank", "Urchif", 19, 'M', false);

if (p1.equals(p2))
    System.out.println("p1 and p2 are equal");
else
    System.out.println("p1 and p2 are not equal");
```

Now you should understand how to write an **equals()** method for your own objects.

The **equals()** method is often called automatically by JAVA when you are searching for something in a collection. For example, consider searching for a particular **Person** in an **ArrayList** of **Person** objects. Assume that we wrote a method that takes an **ArrayList<Person>** as well as a particular **Person** to search for within the list. The method may look as follows:

```
boolean contains(ArrayList<Person> people, Person personToLookFor) {
    for (Person  p: people) {
       if (p.equals(personToLookFor))
            return true;
    }
    return false;
}
```

Notice that the code makes use of the **equals()** method to find a **Person** object in the list that matches the one we are looking for. As it turns out, the standard **contains()** method for **ArrayLists** does exactly this. Hence, for example, if we wanted to write a method that added a particular person **p** to the **people** list, as long as **p** was not already there, we could do it as follows:

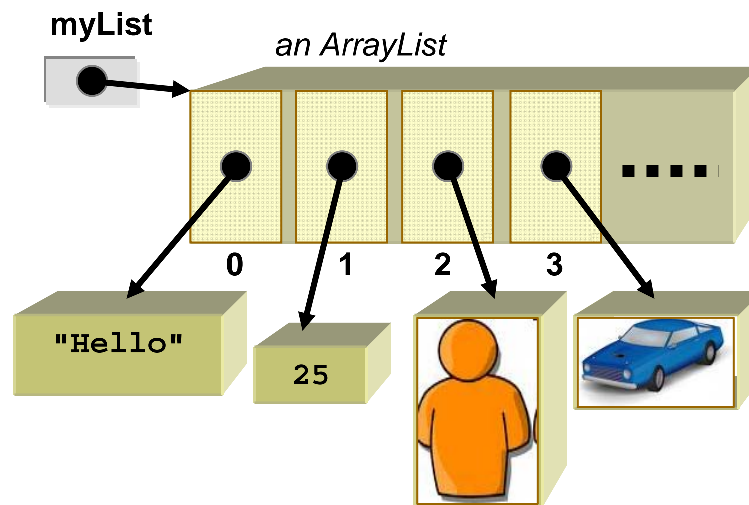
```
void add(ArrayList<Person> people, Person p) {
    if (!people.contains(p))
        people.add(p);
}
```

If we do not write an **equals()** method in the **Person** class (i.e., we inherit the default one from **Object**), then the above method will allow multiple people to be added to the list with the same attribute values.

However, if we write our own **equals()** method, then no two people in the **people** list will have the same attributes … they will all be unique. Hence, it is important for you to understand that other methods (such as **contains()**) will make use of the **equals()** method, whether or not you created one. The results will be different.

10.5 Arrays

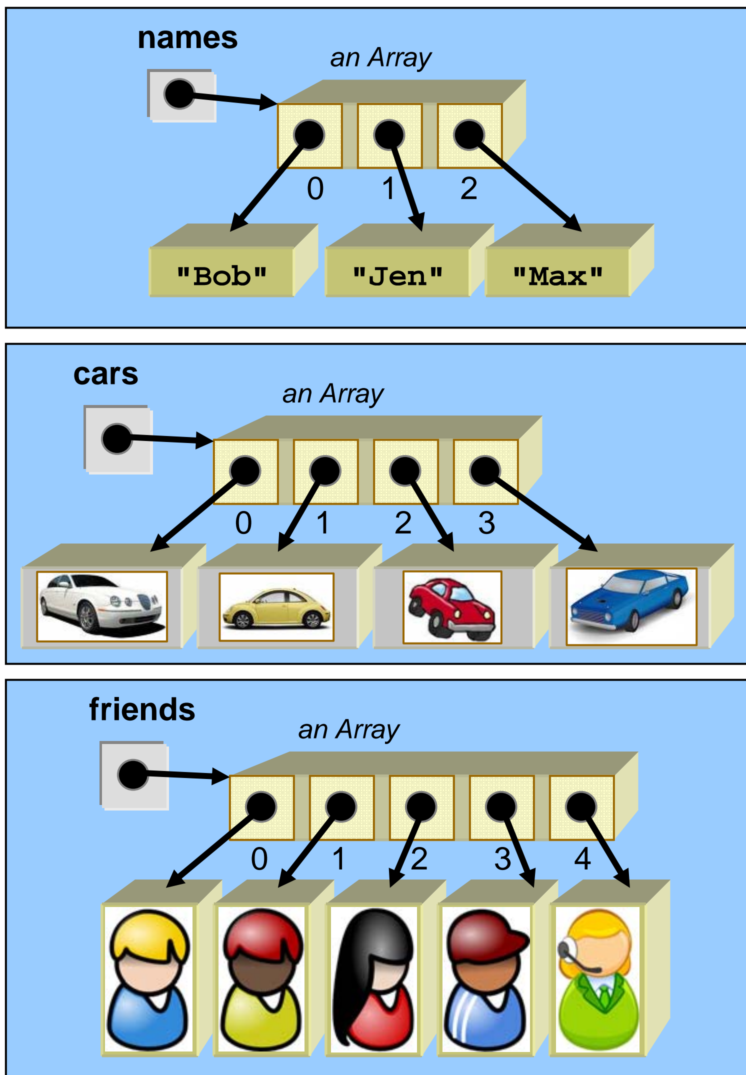
We have already discussed a *collection* class called an **ArrayList** which is capable of storing multiple objects. The objects are all stored in order and we can access or modify the contents of the list through various methods by referring to the objects inside it or by their integer index location.



In JAVA, there is a more restrictive kind of list called an ***Array***. In general, arrays are more efficient in terms of run-time usage, but they are less flexible in the way that they are used. We will discuss here the similarities and differences between arrays and **ArrayLists**.

Like an **ArrayList**, an array is a collection of elements (i.e., data types or objects) which are stored in a particular order and accessible by their index. However, unlike **ArrayLists**, the elements of an ***array*** must ALL be of the ***same type*** (e.g., all Strings, **ints**, Person objects, Car objects, etc…) That is, all the elements must have a common ***type***. In real life, an array could represent a grouped collection of similar "things" such as:

- a list of names
- a list of grades
- a set of hockey cards
- a shelf of books
- etc..



In the pictures to the left, there are three array variables called **names**, **cars** and **friends**. Array variables are declared just like any other variable, except that we need to include square brackets **[ ]** alongside the type as follows:

```
String[]        names;
Car[]           cars;
Person[]        friends;
```

The type can either be an object (as the three above variables show) or a data type. So we can make arrays of **ints**, **double**, **booleans** etc..:

```
int[]           sickDays;
double[]        weights;
boolean[]       retired;
```

Alternatively, we could put the brackets after the variable's name:

```
int           sickDays[];
double        weights[];
boolean       retired[];
String        names[];
Car           cars[];
Person        friends[];
```

All of the above 6 variables here are *object variables* in that they all point to an **Array** object. Looking at the code above, it may seem like **sickDays**, **weights** and **retired** each store "a data type", but actually they each store "an array of data types". So the following two lines of code have very different meanings:

```
int      sickDays;  // variable holds a single int primitive
int[]    sickDays;  // variable holds an array object that contains ints
```

Now these array variable declarations simply reserve space to store array objects, but it actually does not create the array object. So, the following code would print out **null** because the variable is not yet initialized:

```
int[]   sickDays;
System.out.println(sickDays);
```

Notice above that we did not use the square brackets **[]** when you are *using* the array variable in your code ... you only use the brackets when we *define* the variable.

To use this variable (which is just like any other variable), we must give it a value. What kind of value should it have ? An array of course. But it may surprise you to know that we do not call a constructor to create Array objects. Instead, we use a special (similar) syntax as follows:

```
new Person[10]
```

This will create an array that can hold up to 10 **Person** objects. Unlike **ArrayLists**, arrays are fixed size. That means we cannot enlarge them later. So if you are unsure how many items we want to put into the array, you should chose an over-estimate (i.e., a maximum bound) for its size. Here are some examples of how to give a value to our 6 array variables by creating some fixed-size arrays:

```
int[]           sickDays;
double[]        weights;
String[]        names;
Car[]           cars;
Person[]        friends;

sickDays = new int[30];    // creates array that can hold 30 ints
weights = new double[100];// creates array that can hold 100 doubles
names = new String[3];     // creates array that can hold 3 String objects
rentals = new Car[500];    // creates array that can hold 500 Car objects
friends = new Person[50]; // creates array that can hold 50 Person objects
```

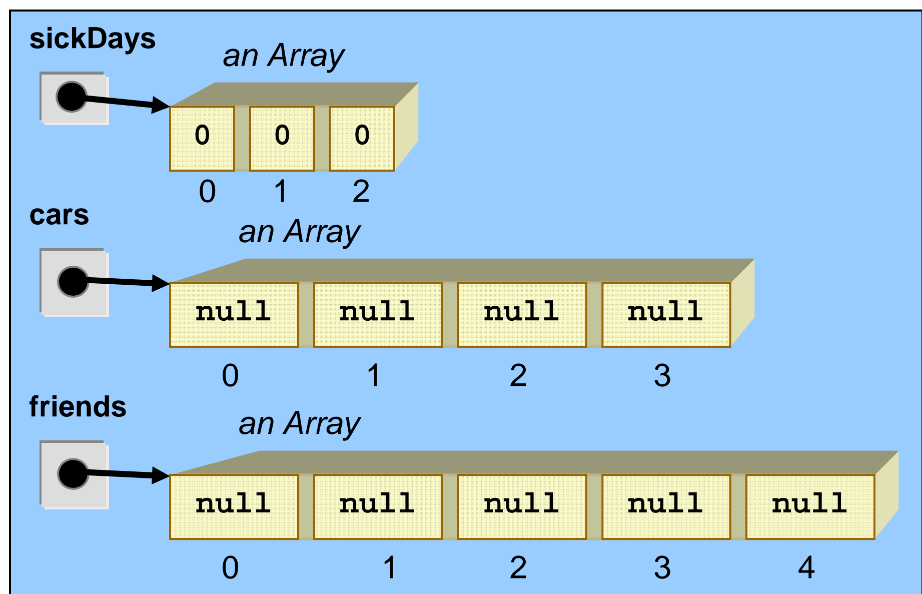
Once we create arrays using the code above, the arrays themselves simply reserve enough space to hold the number of objects (or data types) that you specified. However, it does not create any of those objects!! Newly created arrays are filled with:

- 0 for numbered arrays
- character 0 (i.e., the **null** character) for **char** arrays
- **false** for **boolean** arrays
- **null** for Object-type arrays

So, when you create an array to hold objects, only the array itself is created. The array is NOT initialized with new objects in each location.

Hence, the following code produces the result shown in the picture →

```
int[]        sickDays;
Car[]        cars;
Person[]     friends;

sickDays = new int[3];
cars = new Car[4];
friends = new Person[5];
```



Notice that the arrays which hold object types are simply filled with **null** values … there are no **Car** objects nor **Person** objects created in the above code.

At any time, if we would like to ask an array how big it is (i.e., its size or capacity), we can access one of its special attributes called **length** as follows:

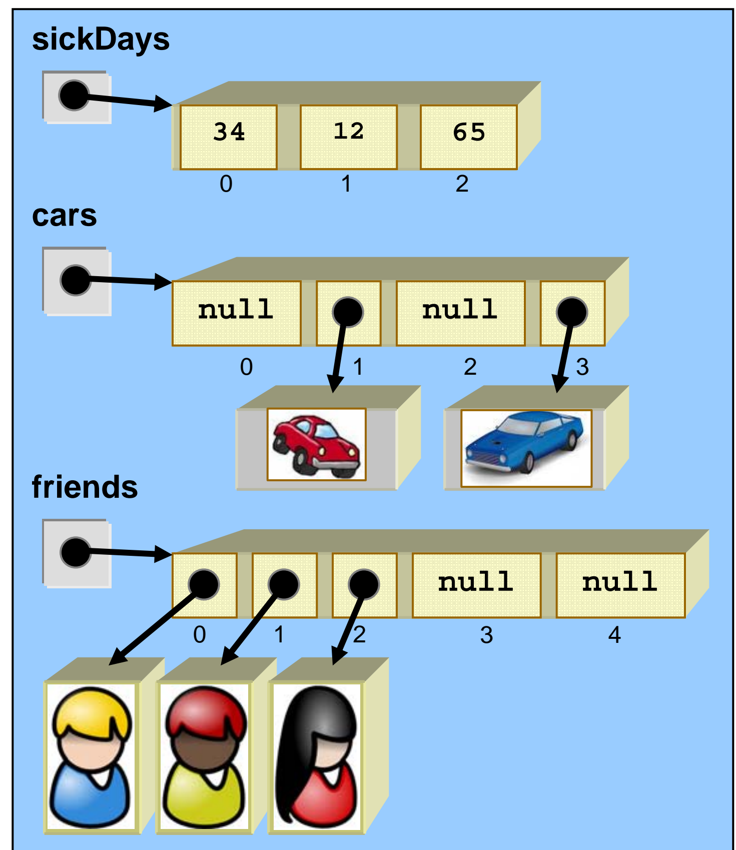
```
System.out.println(sickDays.length);       // displays 3
System.out.println(cars.length);           // displays 4
System.out.println(friends.length);        // displays 5
```

Note that the **length** attribute is NOT a method (i.e., notice that there are no brackets). Also, the **length** attribute ALWAYS returns the exact same size value for the array regardless of the number of elements that you put into the array.

How do we put things into the array ? Values are actually assigned to an array using the **=** operator just as with other variables. However, since the array holds many objects together, we must also specify the location in the array that we want to access or modify. As with **ArrayLists**, the index numbering starts with **0** for the first item and **(length -1)** is the index of the last item.

Here is an example of how to fill in our arrays:

```
int[]       sickDays;
Car[]       cars;
Person[]    friends;

sickDays = new int[3];
cars = new Car[4];
friends = new Person[5];

sickDays[0] = 34;
sickDays[1] = 12;
sickDays[2] = 65;
cars[1] = new Car("Red");
cars[3] = new Car("Blue");
friends[0] = new Person(...);
friends[1] = new Person(...);
friends[2] = new Person(...);
```



The picture to the right shows the result from this code. Notice that we can insert an object at any location in the array, provided that the index is valid.

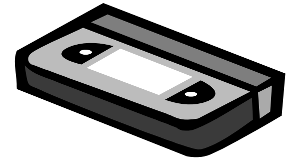
The following two lines of code would produce an **ArrayIndexOutOfBoundsException**:

```
sickDays[3] = 87;              // Error: index 3 is out of range
cars[10] = new Car("Yellow");  // Error: index 10 is out of range
```

A very common mistake made when learning to use arrays is to declare the array variable, but forget to create the array and then try to use it. For example, look at this code:

```
Person[]  friends = new Person[100];
System.out.println(friends[0].firstName);   // Error!
System.out.println(friends[0].age);         // Error!
System.out.println(friends[0].gender);      // Error!
```

Although the above code does *create* an array that can *hold* **Person** objects, it actually never creates any **Person** objects. Hence, the array is filled with 100 values of **null**. The code will produce a **NullPointerException** because **friends[0]** is **null** here and we are trying to access the attributes of a **null** object. We are really doing this: **null.firstName** … which makes no sense.

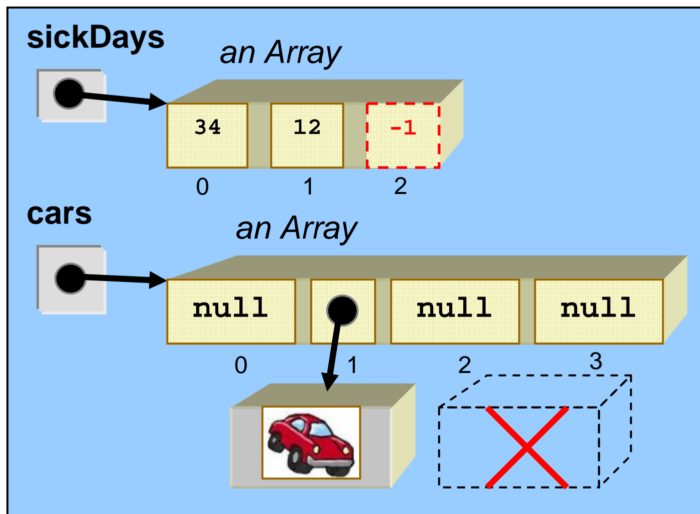


How do we remove something from an array ? Well, as with a program recorded on a videotape, we cannot actually remove the item but we can overwrite it (i.e., replace it) with a new value.

So, to delete information from an array, you usually replace its value with **0** or **null**. The array will stay the same size, but the data will be deleted. For example, to remove the sick day at position **2** in **sickDays** from our previous example, we might replace it with **0**, **-1**, or anything else that tells us that the data is invalid. To remove the blue car from the **cars** array, we would replace it with **null**.

```
sickDays[2] = -1;
cars[3] = null;
```

Here is the result:



What happened to the blue car ? In the array, the **Car** object has been replaced by **null**, and so the array is no longer pointing to the **Car** object. Therefore, the **Car** object is garbage collected (unless some other object is still pointing to it).

Lets get back to adding items to an array. Since arrays are of fixed size and cannot grow, what do we do if we run out of room ? For example, the code below will keep asking for integers from the user, adding them to the array as they come in until **-1** is entered:

```
int         aNumber = 0;
int         currentIndex = 0;
int[]       numbers = new int[5];
Scanner     keyboard = new Scanner(System.in);

// Get contents for the array, one number at a time until -1 entered
while(aNumber != -1) {
    aNumber = keyboard.nextInt();
    if (aNumber != -1)
        numbers[currentIndex++] = aNumber;
}

// Now print the array out
for(int i=0; i<currentIndex; i++)
    System.out.println(numbers[i]+", ");
```

Notice that each number coming in is added to the array according to the **currentIndex** position, which is incremented each time a valid number arrives. When the loop has completed, **currentIndex** represents the number of integers that were entered and added to the array. Can you foresee any problems with the above code ?

When attempting to add the 6th number to the array, an **ArrayIndexOutOfBoundsException** will occur, because the array has only been declared to hold **5** integers. The array has filled up and we must either stop adding to it or make more room for the new numbers.

However, we cannot make more room with the existing array. Rather, a new *bigger* array must be created and all elements must be copied into the new array. But how much bigger ? Its up to us.

Here is how we could change the code to increase the array size by 5 each time it gets full …

```
int         aNumber = 0;
int         currentIndex = 0;
int[]       numbers = new int[5];
Scanner     keyboard = new Scanner(System.in);

// Get contents for the array, one number at a time until -1 entered
while(aNumber != -1) {
    aNumber = keyboard.nextInt();
    if (aNumber != -1) {
        // If at max capacity, make new array and copy contents over
        if (currentIndex == numbers.length) {
            int[] temp = new int[numbers.length + 5];
            for(int i=0; i<numbers.length; i++)
                temp[i] = numbers[i];
            numbers = temp;
        }
        numbers[currentIndex++] = aNumber;
    }
}
// Now print the array out
for(int i=0; i<currentIndex; i++)
    System.out.println(numbers[i]+", ");
```

Of course, we can increase by any amount each time, perhaps even doubling the size. In a similar situation, it may be that *less* space is needed in the array, for example if elements are removed from the array and we don't need as much space anymore. In this case, we could similarly copy the elements over into a *smaller* array and discard the original array.

Assigning individual values to an array like this can be quite tedious, especially when the array is large. Sometimes, in fact, we already know the numbers that we want to place in an array (e.g., we are using some fixed table or matrix of data that is pre-defined). In this case, to save on coding time JAVA allows us to assign values to an array at the time that we create it. This is done by using braces **{ }**. In this case, neither the **new** keyword, the **type** nor the **size** of the array are specified. Instead, we supply the values on the same line as the declaration of the variable. Here are some examples:

```
int[]       ages = {34, 12, 45};
double[]    weights = {4.5, 23.6, 84.124, 78.2, 61.5};
boolean[]   retired = {true, false, false, true};
String[]    names = {"Bill","Jennifer","Joe"};
char[]      vowels = {'a', 'e', 'i', 'o', 'u'};
```

Here, the array's size is automatically determined by the number of values that you specify within the braces, each value separated by a comma. So the sizes of the arrays above are 3, 5, 4, 3 and 5, respectively.

Objects may also be created and assigned during this initialization process as follows …

```
Person[]   people = {new Person("Hank", "Urchif", 19, 'M', false),
                     new Person("Don", "Beefordusk", 23, 'M', false),
                     new Person("Holly", "Day", 67, 'F', true),
                     null,
                     null};
```

Here we are actually creating three specific **Person** objects and inserting them into the first three positions in the **people** array. Notice that you can even supply a value of **null**, for example if you wish to leave some extra space at the end for future information. Hence the people array above has a capacity (i.e., length) of **5**.

As an example, here is an interesting method that uses an array to convert an integer representing a day of the week into the name of the day of the week. For example, the 3rd day of the week is **Tuesday** (assuming that we start the week on a **Sunday**):

```
public static String dayOfWeek(int dayNumber) {
    String days[] = {"Sunday", "Monday", "Tuesday", "Wednesday",
                     "Thursday", "Friday", "Saturday"};
    if ((dayNumber < 1) || (dayNumber > 7))
        return "Invalid";

    return days[dayNumber-1];
}
```

Do you understand why we are subtracting 1 from the **dayNumber** ? I hope so. To use this method, we would have to write it in some class and then call it ... although the details have been left out here.

Lets look at some simple examples that use arrays to make sure that you understand how to use them. First of all, given an array of integers, here is a program to find their average:

```
int  nums[] = {23, 54, 88, 98, 23, 54, 7, 72, 35, 22};
int  sum = 0;

for (int i=0; i<nums.length; i++)
    sum += nums[i];

System.out.println("The average is " + sum/(double)nums.length);
```

Recall that we need to divide by a **float** or **double** in order to get a more accurate result, hence the type cast to **double** here. If we would have left out the type cast, the result would have discarded the fractional portion of the computation, giving only a truncated integer result.

What if we were to create a method to compute the average of an *arbitrary* array ? We would need to place this code in some class, perhaps called **ArrayCalculator** as follows …

```
public class ArrayCalculator {
    static double calculateAverage(int[] nums) {
        int sum = 0;
        for (int i=0; i<nums.length; i++)
            sum += nums[i];
        return sum/(double)nums.length;
    }
}
```

Notice that the method takes an **int** array as a parameter and returns a double as the result. Here is a test program that we can use to try out the method:

```
public class ArrayCalculatorTestProgram {
    public static void main(String args[]) {
        int ages[] = {23, 54, 88, 98, 23, 54, 7, 72, 35, 22};
        System.out.print("The average is ");
        System.out.println(ArrayCalculator.calculateAverage(ages));
    }
}
```

Notice that when passing an array as a parameter, we DO NOT use the square brackets:

```
arrayCalc.calculateAverage(ages[]);      // WRONG!!
```

and we DO NOT specify the type either:

```
arrayCalc.calculateAverage(int ages[]); // WRONG!!
```

Here is another example. Given an array of 10 numbers, how do we write a method that can find the maximum ? The method should take in an **int** array parameter and return an **int** as the result:

```
public static int findMaximum(int[] nums) {
    int max = 0;
    for (int i=0; i<nums.length; i++) {
        if (nums[i] > max)
            max = nums[i];
    }
    return max;
}
```

Notice that the code goes through the numbers in the array and simply replaces the value of the **max** variable with any number that it finds to be larger. It is important to note that the return must occur *after* the loop, that is, after you have checked all of the numbers. A common mistake by students is to put the return in the **IF** statement (as shown below) but this would not work. Do you know why ?

```
public static int findMaximum(int[] nums) {
    int max = 0;
    for (int i=0; i<nums.length; i++) {
        if (nums[i] > max)
            return nums[i];    // WRONG!
    }
}
```

In the above example, the initial value for **max** was set to **0**. Could you foresee any situations in which this could pose a problem ?

Well, if all of the integers are negative, then the code will return 0 as the maximum, which is an error. If however, you are certain that at least one number is positive or 0, then the code will work fine. However, a safer way to do this is to set the value of max initially to `Integer.MIN_VALUE`, which is a constant which represents the smallest **int** that can be stored in JAVA. Another option is to set **max** to be the first element in the array (i.e., **nums[0]**), which would always produce correct results as well.

Let us write one more method. This time we will write one that modifies the elements of an array. We will write a method, called **scale()** that will take an array of integers and multiply them by the some scale factor, which will also be passed as a parameter.

```
public static void scale(int[] nums, int factor) {
    for (int i=0; i<nums.length; i++)
        nums[i] = nums[i] * factor;
}
```

The method is quite straight forward. Each element of the array is simply replaced by that same element multiplied by the factor. As a result, the original array that was passed in would now be changed. We can actually simplify the code a little by making use of the “for each” loop that we used with **ArrayLists**. Here is how we could use it on the **calculateAverage()** and **findMaximum()** methods …

```
public static double calculateAverage(int[] nums) {
    int sum = 0;
    for (int i:  nums)
        sum += i;
    return sum/(double)nums.length;
}

public static int findMaximum(int[] nums) {
    int max = 0;
    for (int i: nums) {
        if (i > max)
            max = i;
    }
    return max;
}
```

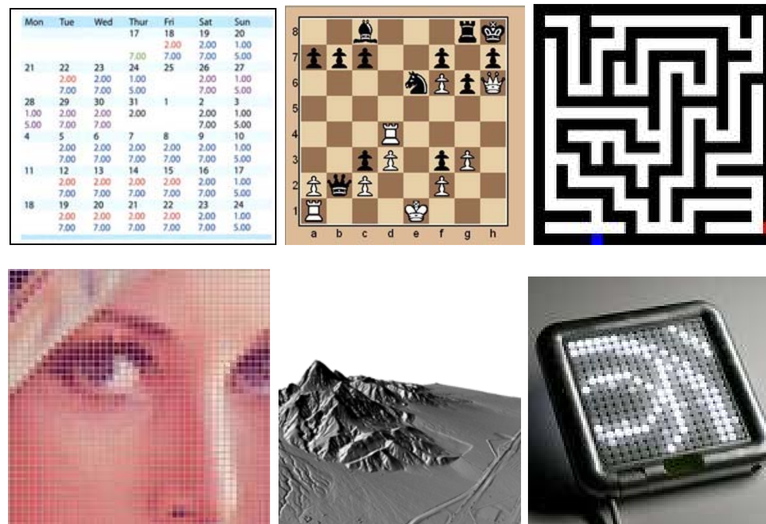
Notice that **i** now represents the number from the array, NOT its index! Interestingly, we cannot use the "for each" loop for the **scale()** method:

```
static void scale(int[] nums, int factor) {
    for (int i: nums)
        i = i * factor;     // WRONG!
}
```

Why not ? This will not work because **i** is a temporary loop variable that points to the item in the array. The code above simply re-assigns the value of **i * factor** to the loop variable **i**, but does not affect the element from the array. So, we would need to use the regular "for" loop where **i** represents the index, so that we can do **nums[i] = nums[i] * factor;**

Multi-Dimensional Arrays

JAVA allows arrays of multiple dimensions (2 or more). 2-dimensional (i.e., 2D) arrays are often used to represent data tables, game boards, mazes, pictures, terrains, displays etc…



In these cases, the information in the array is arranged by ***rows*** and ***columns***. Hence, to access or modify something in the table/grid, we need to specify which row and column the item lies in. Therefore, instead of using just one index as with simple arrays, a 2D array requires that we always supply two indices … row and column.

Therefore, in JAVA, we specify a 2D array by using two sets of square brackets **[ ] [ ]**. Therefore, our variables should have both sets of brackets when we declare them:

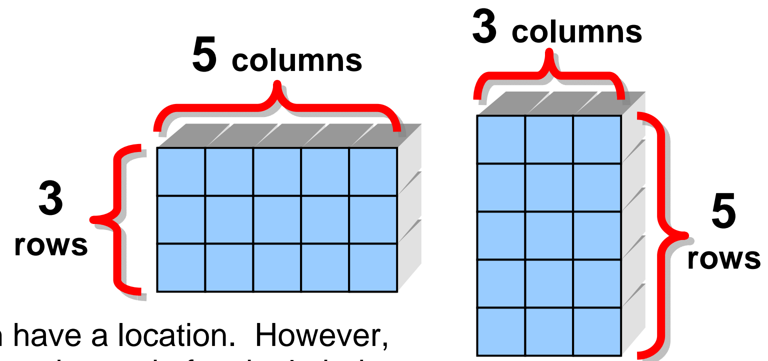
```
int[][]     schedule;   // a 2D array
int[]       list;       // a 1D array
int         age;        // not an array, just an int
```

Also, when creating the arrays, we must specify the number of rows as well as the number of columns. Remember, the arrays cannot grow, so these should represent the maximum number of rows and columns that we want to have:

```
schedule = new int[10][10];       // table with 10 rows and 10 columns
image = new Pixel[1024][768];     // a 1024x768 pixel image
parkingLot = new Car[12][4];      // 12x4 lot that can hold 48 cars
```

Usually, intuitively, the first length given represents the number of rows while the second represents the number of columns, but this need not be the case. For example, the following line of code creates an array that can hold 15 items:

```
grid1 = new int[3][5];
```

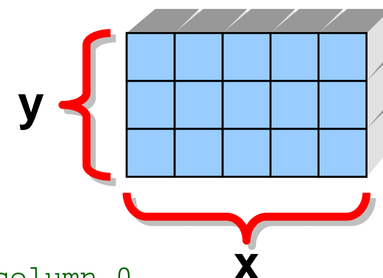


You can think of it as being either a **3x5** array or a **5x3** array. It is up to you as to whether the **3** is the rows or the **5** is the rows. You just need to make sure that you are consistent.

As with regular arrays, the elements each have a location. However, for 2D arrays, the location is a pair of indices instead of a single index. The rows and columns are all numbered starting with 0. Therefore, we access and modify elements from the array by specifying both row and column indices as follows:

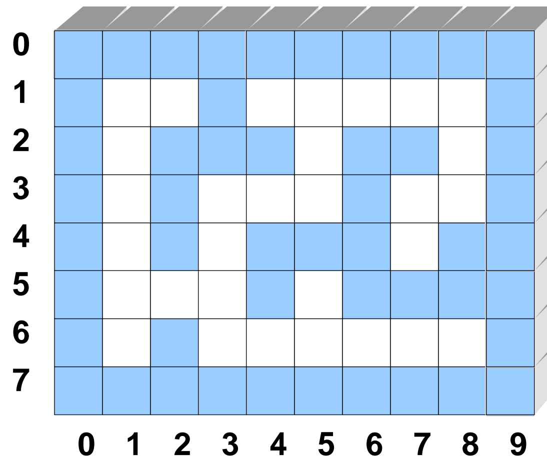
```
schedule[0][0] = 34;    // row 0, column 0
schedule[0][1] = 15;    // row 0, column 1
schedule[1][3] = 26;    // row 1, column 3
```

Sometimes, there is confusion, for example, when we create grids with **(x,y)** coordinates because when dealing with coordinates we always specify **x** before **y**. But visually, **x** represents the number of *columns* in a grid, while **y** represents the number of *rows* … hence **(x,y)** corresponds to **(columns,rows)** which seems counter-intuitive.




```
points[0][0] = 34;    // (x,y)=(0,0) = row 0, column 0
points[0][1] = 15;    // (x,y)=(0,1) = row 1, column 0
points[1][3] = 26;    // (x,y)=(1,3) = row 3, column 1
```

As an example that uses 2D arrays, let us write code that will create and display the following maze:



We could represent this maze by using an array of **ints** indicating whether or not there is a wall at each location in the array (i.e., 1 for wall, 0 for open space). Notice how we can do this by using the quick array declaration with the braces **{ }** as we did with 1D arrays:

```
int[][]      maze = {{1,1,1,1,1,1,1,1,1,1},
                     {1,0,0,1,0,0,0,0,0,1},
                     {1,0,1,1,1,0,1,1,0,1},
                     {1,0,1,0,0,0,1,0,0,1},
                     {1,0,1,0,1,1,1,0,1,1},
                     {1,0,0,0,1,0,1,1,1,1},
                     {1,0,1,0,0,0,0,0,0,1},
                     {1,1,1,1,1,1,1,1,1,1}};
```

Notice that there are more brace characters than with 1D arrays. Each row is specified by its own unique braces and each row is separated by a comma. In fact, each row is itself a 1D array. Interestingly, the **length** field of a multi-dimensional array returns the length of the first dimension only. Consider this code:

```
int[][] ages = new int[4][7];
System.out.println(ages.length);       // displays 4, not 28!
```

In fact, we can actually access the separate arrays for each dimension:

```
int[][] ages = new int[4][7];
int[] firstArray = ages[0];            // gets 1st row from ages array
System.out.println(ages.length * firstArray.length);  // displays 28
```

Therefore, as you can see, a 2D array is actually an array of 1D arrays.

Iterating through a 2D array is similar to a 1D array except that we usually use 2 nested **for** loops. Here is some code to print out the maze that we created above:

```
for (int row=0; row<8; row++) {
    for (int col=0; col<10; col++)
        System.out.print(maze[row][col]);
```

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

And here would be the result:

```
1111111111
1001000001
1011101101
1010001001
1010111011
1000101111
1010000001
1111111111
```

Of course, we can make the maze look however we want. Here is how we can display a different wall character. Notice how we switched the loops (just for fun) to use the “for each” style:

```
    for (int[] row:  maze) {
        for (int item:  row) {
            if (item == 1)
                System.out.print('*');
            else
                System.out.print(' ');
        }
        System.out.println();
    }
```

And here would be the result:

```
**********
*  *     *
* *** ** *
* *   *  *
* * *** **
*   * ****
* *      *
**********
```

PRACTICE: Assuming that you set one location in the array to 2 and another location to 3, could you write a program that would determine how long it would take (i.e., steps) to get from 2 to 3 ?

	0	1	2	3	4	5	6	7	8	9
0	1	1	1	1	1	1	1	1	1	1
1	1	2	0	1	0	0	0	0	0	1
2	1	0	1	1	1	0	1	1	0	1
3	1	0	1	0	0	0	1	0	0	1
4	1	0	1	0	1	1	1	3	1	1
5	1	0	0	0	1	0	1	1	1	1
6	1	0	1	0	0	0	0	0	0	1
7	1	1	1	1	1	1	1	1	1	1

Interestingly, you can create even higher dimensional arrays. For example, an array as follows may be used to represent a cube of colored blocks:

```
int cube[][][] = new Color[3][3][3];
```

Notice that there are now 3 sets of square brackets. Using 3D arrays works the same way as with 2D arrays except that we now use 3 sets of brackets and 3 indices when referring to the elements of the array.

3-dimensional arrays are often used in the real world to model various objects: