Variables and Memory

- Variables are stored in main memory at specific addresses that are determined when the program enters the method.
- Variables of different types require different number of bytes to hold their data.
Variables and Memory: An Example

```java
public static void main(String[] args) {
    int x, y;
    byte a
    ...
}
```

Objects and References

- When an object is declared, space is set aside to hold the address at which the object’s data is stored, e.g.,

  ```java
  MyClass myObject;
  ```

- There is a location in memory where the address is stored at which all the data about `myObject` is contained. The address that is stored is called a `reference`. 
Declaring an Object

MyClass myObject;

null

Constructing an Object

MyClass myObject
    = new MyClass();

Data About myObject
ToyClass.java

public class ToyClass {
    private String name;
    private int number;

    public ToyClass(String initialName,
            int initialNumber) {
        name = initialName;
        number = initialNumber;
    }

    public ToyClass() {
        name = "No name yet";
        number = 0;
    }

    public void set(String newName, int newNumber) {
        name = newName;
        number = newNumber;
    }

    public String toString() {
        return (name + " " + number);
    }

    public static void changer(ToyClass aParameter) {
        aParameter.name = "Hot Shot";
        aParameter.number = 42;
    }

    public boolean equals(ToyClass anotherObject) {
        return (name.equals(anotherObject.name)
                && number == anotherObject.number);
    }
}
public class TestToyClass {
    public static void main(String[] args) {
        ToyClass variable1 = new ToyClass("Joe", 42);
        ToyClass variable2;
        variable2 = variable1;

        // Now both variables name the same object.
        variable2.set("Josephine", 1);
        System.out.println(variable1);
    }
}

Output
Josephine 1
Tracing through **TestToyClass** (continued)

```java
variable2 = variable1;
```

- `variable1`:
  - 4068
    - "Joe"
  - 4068
  - ...

- `variable2`:
  - 4068

---

Tracing through **TestToyClass** (continued)

```java
variable2.set("Josephine", 1);
```

- `variable1`:
  - 4068
    - "Josephine"
  - 4068
  - ...

- `variable2`:
  - 4068
Objects As Parameters

- All parameters are passed by value, i.e., the value gets copied over into the method’s memory area.
- For objects, it is the reference to the object that gets copied over, i.e., the address at which the object’s data appears. Any changes made to an object passed as a parameter are permanent; the calling procedure knows about them.

ClassParameterDemo.java – An Example

```java
public class ClassParameterDemo {
    public static void main(String[] args) {
        ToyClass anObject = new ToyClass("Mr. Cellophane", 0);
        System.out.println(anObject);
        System.out.println("Now we call changer with anObject as argument.");
        ToyClass.changer(anObject);
        System.out.println(anObject);
    }
}
```

Output
Mr. Cellophane 0
Now we call changer with anObject as argument.
Hot Shot 42
Tracing through `ClassParameterDemo`

Before

```
<table>
<thead>
<tr>
<th>aParameter</th>
<th>3078</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>anObject</td>
<td>3078</td>
</tr>
</tbody>
</table>
```

Data About anObject

```
"Mr. Cellophane"
```

```
0
```

Tracing through `ClassParameterDemo` (continued)

```
ToyClass.changer(anObject);
```

```
<table>
<thead>
<tr>
<th>aParameter</th>
<th>3078</th>
</tr>
</thead>
<tbody>
<tr>
<td>3078</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
"Mr. Cellophane"
```

```
0
```

Data About anObject
Tracing through `ClassParameterDemo` (continued)

```java
aParameter.name = "Hot Shot";
aParameter.number = 42
```

Comparing Objects

- To test the equality of two objects, it is always a good idea to compare them using a `equals` method.
- When you compare two objects using the `==` operator, you are comparing the two references.
null

- **null** is a special constant that can be assigned to any object reference, indicating that it has no “real” value.
- It is also used to indicate when an object has not yet been constructed.

```java
YourClass yourObject = null;
...
if (yourObject == null)
    System.out.println
        ("No real object here.");
```
Anonymous Objects

- When I write
  `ToyClass variable1 = new ToyClass("Joe", 42);`
  I am calling a constructor to create an instance of object in memory whose reference is stored in variable 1.
- I can write
  ```java
  if (variable1.equal(new ToyClass("JOE", 42)))
      System.out.println("Equal");
  else
      System.out.println("Not equal");
  ```
- Since I will NEVER want to use `new ToyClass("JOE", 42)` again. This is an anonymous object; it has no name to identify it.

Anonymous Objects Are Temporary

- Writing
  ```java
  if (variable1.equal(new ToyClass("JOE", 42)))
      System.out.println("Equal");
  else
      System.out.println("Not equal");
  ```
  is equivalent to
  ```java
  ToyClass temp = new ToyClass("JOE", 42);
  if (variable1.equal(temp))
      System.out.println("Equal");
  else
      System.out.println("Not equal");
  ```
Invariants

- An invariant is a statement that is always true. E.g,
  \[
  \text{(for } i = 0; \ i < 5 \ \text{i++)} \ \{
  ...
  ...
  \}
  \// \text{Invariant } i \geq 5
  \]
- A class invariant is a statement that is always true about a class.

Person.java

// Class for a person with a name and dates for
// birth and death.
// Class invariant - A Person always has a date
// of birth and if there is a
// date of birth it precede the
de date of birth
public class Person {
    private String name;
    private Date born;
    private Date died; // null indicates still
    // alive
```java
// Person() - This constructor initializes the
//            Date objects uses their copy
//            constructor.
public Person(String initName,
               Date birthDate, Date deathDate) {
    if (consistent(birthDate, deathDate)) {
        name = initName;
        born = new Date(birthDate);
        if (deathDate == null)
            died = null;
        else
            died = new Date(deathDate);
    } else {
        System.out.println
            ("Inconsistent dates. Aborting.");
        System.exit(0);
    }
}

// Person() - A copy constructor for Person
public Person(Person original) {
    // It is essential that we determine if the
    // original object exists by seeing if its
    // reference is null or not.
    if (original == null) {
        System.out.println("Fatal error.");
        System.exit(0);
    }
    name = original.name;
    born = new Date(original.born);

    // If we had written born = original.born
    // it would be an example of a privacy leak
    if (original.died == null) // Not a problem
        died = null;
    else
        died = new Date(original.died);
}
```
// Mutator that sets the values for the whole class
public void set(String newName, Date birthDate, Date deathDate) {
    // Make sure that the dates are consistent
    if (consistent(birthDate, deathDate)) {
        name = newName;
        // born cannot be null but died might be
        born = new Date(birthDate);
        if (died == null)
            died = new Date(deathDate);
        else
            died = deathDate;
    }
}

// toString() - converts a person's data into a String
public String toString() {
    // First, let's see if there is a date of death
    String diedString;
    if (died == null)
        diedString = ""; // Empty string
    else
        diedString = died.toString();

    // born is converted using Date.toString()
    return(name + ", " + born
            + "-" + diedString);
}
public boolean equals(Person otherPerson) {
    if (otherPerson == null)
        return false;
    else
        return (name.equals(otherPerson.name)
                && born.equals(otherPerson.born)
                && datesMatch(died, otherPerson.died));
}

private static boolean datesMatch(Date date1, Date date2) {
    if (date1 == null)
        return (date2 == null);
    else if (date2 == null) // and date1 != null
        return false;
    else // both dates are NOT null
        return (date1.equals(date2));
}
// setBirthDate(Date newDate) - A mutator for born
//   Precondition: newDate is a consistent date of birth
//   Postcondition: Date of birth of the calling object is newDate
public void setBirthDate(Date newDate) {
    if (consistent(newDate, died))
        born = new Date(newDate);
    else {
        System.out.println
            ("Inconsistent dates. Aborting.");
        System.exit(0);
    }
}

// setDeathDate() - A mutator for deathDate
public void setDeathDate(Date newDate) {
    if (!consistent(born, newDate)) {
        System.out.println
            ("Inconsistent dates. Aborting.");
        System.exit(0);
    }

    // A null date of death is valid. This is different from setBirthDate because
    // birthDate cannot be null.
    if (newDate == null)
        died = null;
    else
        died = new Date(newDate);
}
// setName() - A mutator for name
//   Precondition: name cannot be null
public void setName(String newName) {
    name = newName;
}

// getName() - An accessor for name
public String getName() {
    return name;
}

// getBirthDate() - An accessor for birthDate
public Date getBirthDate() {
    return new Date(born);
}

// setBirthYear - A mutator for the person's
//   year of birth
//   Precondition: the date of birth has been
//   set and changing the year will give
//   consistent date of birth.
//   Postcondition: The year of birth is newYear
public void setBirthYear(int newYear) {
    if (born == null) {
        // Precondition is violated
        System.out.println
            ("Fatal error. Aborting.");
        System.exit(0);
    }
    born.setYear(newYear);
    if (!consistent(born, died)) {
        System.out.println
            ("Inconsistent dates. Aborting.");
        System.exit(0);
    }
}
// setDeathYear - A mutator for the person's
//                year of death
// Precondition: the date of death has been
//        set and changing the year will give
//        consistent date of death.
// Postcondition: The year of death is newYear
public void setDeathYear(int newYear) {
    if (died == null) {
        // Precondition is violated
        System.out.println
            ("Fatal error. Aborting.");
        System.exit(0);
    }
    died.setYear(newYear);
    if (!consistent(born, died)) {
        System.out.println
            ("Inconsistent dates. Aborting.");
        System.exit(0);
    }
}

// getDeathDate() - An accessor for deathDate
// We must ensure that it
// exists; else return null
public Date getDeathDate() {
    if (died == null)
        return null;
    else
        return new Date(died);
}
// To be consistent birthDate must not be null.
// If there is no date of death
// (deathDate == null), that is consistent with
// any birthDate. Otherwise, the birthDate
// must come before or be equal to the
// deathDate.
private static boolean consistent
    (Date birthDate, Date deathDate) {
    if (birthDate == null)
        return false;
    else if (deathDate == null)
        return true;
    else
        return(birthDate.precedes(deathDate)
            || birthDate.equals(deathDate));
}

PersonDemo.java

public class PersonDemo {
    public static void main(String[] args) {
        Person bach
            = new Person("Johann Sebastian Bach",
                new Date("March", 21, 1685),
                new Date("July", 28, 1750));
        Person stravinsky
            = new Person("Igor Stravinsky",
                new Date("June", 17, 1882),
                new Date("April", 6, 1971));
        Person adams
            = new Person("John Adams",
                new Date("February", 15, 1947), null);
System.out.println
    ("A Short List of Composers:");
System.out.println(bach);
System.out.println(stravinsky);
System.out.println(adams);

Person bachTwin = new Person(bach);
System.out.println
    ("Comparing bach and bachTwin:");
if (bachTwin == bach)
    System.out.println
        ("Same reference for both.");
else
    System.out.println("Distinct copies.");

if (bachTwin.equals(bach))
    System.out.println("Same data.");
else
    System.out.println("Not same data.");

Output
A Short List of Composers:
Johann Sebastian Bach, March 21, 1685–July 28, 1750
Igor Stravinsky, June 17, 1882–April 6, 1971
John Adams, February 15, 1947–
Comparing bach and bachTwin:
Distinct copies.
Same data.
Mutable and Immutables Classes

- **Mutable classes** – classes where properties can be changed, either because they are public or because there are mutators for the properties.
- **Immutable classes** – classes whose properties cannot be changed because they are private and there are no mutator methods.

Mutable and Immutable Classes – An Example

- Consider the two methods from `Person.java`:
  ```java
  public String getName() {
      return name;
  }
  public Date getBirthDate() {
      return new Date(born);
  }
  ```
  - Why do we return a reference to name but create a duplicate object for born? *Because name cannot be changed directly.*
Changing Strings

• When you write:
  
  ```java
  String greeting = "hello";
  greeting = greeting + " friend";
  ```

  we are replacing the reference to the object `greeting` with another
  reference to a new object that replaces the old `greeting`.

• If we write:
  
  ```java
  String greeting = "hello";
  String helloVariable = greeting;
  greeting = greeting + "friend";
  System.out.println(helloVariable);
  System.out.println(greeting);
  ```

  • The output will be
  
  hello
  hello friend

Packages and Import Statements

• A **package** is Java’s way of forming a library of classes.

• If you wish to use a package of classes that are not the directory in which you are working, write:

  ```java
  import java.util.Scanner; // to include Scanner
  Import java.util.*; // to include all of
                      // java.util’s classes.
  ```
Package Names and Directories

- A package can be created by grouping all the classes together in one directory and placing the line
  ```java
  Package PackageName;
  ```
  At the beginning before any Java code.
- Now these classes can be used by adding an import statement to the classes that use it.