Lecture 19

We discuss types.

We distinguish Java's *primitive types* from its *reference types*. We see that variables of these different types hold quite different things and so the effects of assignment and printing can be quite different too.

Annotate your copy of these notes with lots of pictures during the lecture


```
public class Location
{  public Location(int aLongitude,
      int aLatitude)
   {  longitude = aLongitude;
      latitude = aLatitude;
   }

   public int getLongitude()
   {  return longitude;
   }

   public int getLatitude()
   {  return latitude;
   }

   public void move(int aLongitude,
      int aLatitude)
   {  longitude = aLongitude;
      latitude = aLatitude;
   }

   private int longitude;
   private int latitude;
}
```


Strong and static typing

- Java is a *strongly-typed* language:

  Every variable must have a type. Java then uses this to restrict the values a variable can hold, the operators that may be applied to it and the messages it can be the target of.

- Java is (mostly!) a *statically-typed* language:

  The type of every variable and (nearly) every expression is known (from the program text) at compile-time.

  Hence, checking for *type violations* can (mostly!) be done at compile-time.

Primitive types

- A variable may have one of Java's *primitive types*:
  - these types are built-into the language: `byte, short, int, long, float, double, boolean, char`
  - these variables hold values that have direct machine representations

Data type	Values	Default
`byte`	8-bit signed 2's-comp integer	0
`short`	16-bit signed 2's-comp integer	0
`int`	32-bit signed 2's-comp integer	0
`long`	64-bit signed 2's-comp integer	0
`float`	32-bit floating-point number	+0.0
`double`	64-bit floating-point number	+0.0
`boolean`	`true` or `false`	`false`
`char`	16-bit Unicode character	`\u0000`

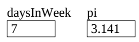
Reference types

- Or, a variable may have a *reference type*:
  - with one exception, they are not built-into the language: each class that a programmer defines gives us one such type, e.g. `String`, `Color`, `BankAccount`, `Fraction`
  - the one that is built-in is arrays
  - to create a value belonging to a reference type, you use `new` (except when exploiting the shortcut for `String`)
  - variables of these types hold references ('pointers') to appropriate objects or `null`

They hold different kinds of things!

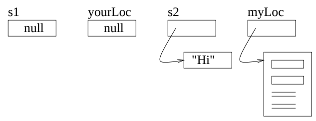
Primitive type variables:

```
int daysInWeek = 7;
double pi = 3.141;
```



Reference type variables:

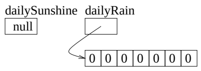
```
String s1 = null;
Location yourLoc = null;
String s2 = "Hi";
Location myLoc = new Location(8, 51);
```



Arrays

- Arrays are the only built-in reference types
- So array variables hold `null` or a reference ('pointer'):

```
int[] dailySunshine = null;
int[] dailyRain = new int[7];
```

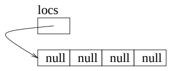


- (A common error is to forget to allocate the array, in which case, when you access the array you get a 'null pointer exception')

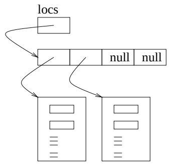
Arrays of reference types

```
Location[] locs = null;
```


```
locs = new Location[4];
```




```
locs[0] = new Location(8, 51);
locs[1] = new Location(10, 52);
```



Printing the contents of variables

- Printing will show the value/reference:

```
int x = 10;
System.out.println(x);
Location loc = new Location(8, 51);
System.out.println(loc);
```

- So, it's a good idea to include a `toString` method in every class definition, e.g.:

```
public String toString()
{  return longitude + ", " + latitude;
}
```

- `println` invokes `toString`. Equivalent:

```
System.out.println(loc.toString());
System.out.println(loc);
```


Assignment copies contents

```
int x = 10;
int y = x;
```

— `x` contains 10, so a copy of this value is put into `y`

```
Location myLoc = new Location(8, 51);
Location yourLoc = myLoc;
```

— `myLoc` contains a reference, so a copy of this reference is put into `yourLoc`, giving two references to the same object (not two objects)

Consequences

Mostly, there are no problems. But sometimes there are. . .

```
Location myLoc = new Location(8, 51);
Location yourLoc = myLoc;
System.out.println(myLoc);
System.out.println(yourLoc);
myLoc.move(10, 52);
System.out.println(myLoc);
System.out.println(yourLoc);
```

— you move too!

To avoid this, ensure there are two distinct objects (not two references to the same object), e.g.:

```
Location yourLoc = new Location(
  myLoc.getLongitude(), myLoc.getLatitude())
```

or include a copy method in `Location` that returns a new object with the same contents:

```
Location yourLoc = myLoc.clone();
```


Primitive types	**Reference types**
Examples	
`int, double, boolean, char`	arrays, `String`, `Color`, `BankAccount`, `Fraction`, `Location`
Contents	
Contains a value	Contains a reference ('pointer') to an object
System.out.println(. . .)	
Displays the value	Runs `toString`. `toString`, by default, displays the pointer. You can supply your own definition to display something more meaningful
Assignment	
Copies the value	Copies the reference (hence two pointers to same object)