Basic Types

- Most languages come with some basic types
  - Numbers of various sorts/ranges
  - Boolean values
  - Characters of text
- Even here, there are some differences, however
  - C was implemented without a specific Boolean type, and simply used an integer value (0 as false, anything else true)

```java
int a = 0;
if (a = 1) {
    printf("true");
}
```

Prints “true”
In Java, e.g., will not compile

Enumerations

- An enumerated type serves to specify both the type itself along with the allowed instances
- Supported by C, Java, and many others

```java
enum Direction {north, south, east, west};
Direction d = Direction.north;
```

- Scala doesn’t have a built-in type, although it can simulate the idea via library support

```java
object WeekDay extends Enumeration {
    val Mon, Tue, Wed, Thu, Fri, Sat, Sun = Value
}
```

Enumerations in Java

- In Java, enumerations possess two methods
  - `ordinal()` returns numeric index of enumerated value
  - `name()` returns the name of the instance as a `String`

```java
enum Direction {north, south, east, west};
Direction d = Direction.north;
int x = d.ordinal(); // 0
String name = d.name(); // "north"
```
Enumerations in C

- In C, enumerations are also really int types

```c
enum {MON, TUE, WED, THU, FRI} weekday;
weekday = MON;
weekday = TUE;
for(weekday = MON; weekday <= FRI; weekday++) {
    printf("%d\n", weekday);
}
weekday = 518; // this is bad but compiles!
```

Beyond Basic Types

- Beyond the elementary types, languages may include a number of composite types, generally built out of a combination of the basic elements of the language
  - Records/structs
  - Arrays
  - Sets
  - Pointers
  - Lists
  - Files
  - ...

Type Conversions

- A type conversion occurs when something of one type is changed to something of a different type
- A narrowing conversion occurs when we either:
  1. Convert some basic type to one that uses fewer bits (or at least has a smaller legal range)
  2. Convert from some more general supertype to a more specific subtype in an object-oriented environment

- When dealing with basic types, a narrowing conversion has the potential to lose information/precision
- When dealing with object-oriented type hierarchies, a successful narrowing gives us more specific information about the objects that result from it

```c
int i1 = 11;
int i2 = (int) i1;
double n = (double) i2;
```

Type Conversions

- Conversions can be handled by coercion or casting
  - In the first case, values are converted to other types, either at compile or run time, without the programmer needing to do anything explicitly to make this happen
  - In the second case, some language construct is explicitly invoked to cause the conversion

- Note that many languages allow explicit casting even where it is not necessary, as in Java:

```c
int i1 = 11;
int i2 = (int) i1;
double n = (double) i2;
```
### Type Coercion

- Real languages have complex rules for coercions.
- Rules for Java conversions for all numeric operators:
  1. If either operand is of type `double`, convert the other to `double` and compute a `double` result.
  2. Otherwise, if either operand is of type `float`, convert the other to `float` and compute a `float` result.
  3. Otherwise, if either operand is of type `long`, convert the other to `long` and compute a `long` result.
  4. Otherwise, convert both operands to `int` and compute an `int` result.
  5. The left-hand side of an assignment must have a type to which the right-hand side can be converted.

```java
double x = 2 * 7 + (8.78 / 1 - 3f);
```

### Coercion is Complicated!

- Coercion rules are often-times one of the most complex parts of a language for inexperienced programmers to understand.
- Some have argued that this is a reason to eliminate coercion from languages altogether, as it is often the source of hard bugs.

```java
double num = Integer.MAX_VALUE + 1;
System.out.printf("num = %f\n", num);
```

**Prints:**
```
num = -2147483648.000000
```

(Coercion happens on assignment, after addition, with integer overflow)

### Casting is Complicated, Too!

- In a language like Java, coercions only occur when they are **widening** conversions: from one basic type to another that has greater range, or to a supertype as needed.
- Narrowing conversions generally must be performed explicitly.

```java
double big = Integer.MAX_VALUE * 2.0;
int i = (int) big;
```

Not legal: can’t convert double to int

```java
double big = Integer.MAX_VALUE * 2.0;
int i = big;
```

Legal: but what is the value of i?

### Casting is Complicated, Too!

- In the Java language specification, the rule on handling a narrowing conversion like this one are that execution:
  1. Truncates the larger `double` value, removing all decimal precision
  2. Assigns the resulting integer portion into the `int` variable, unless the result is too large (or small), in which case it assigns the maximum (or minimum) possible `int` value to that variable.

```java
double big = Integer.MAX_VALUE * 2.0;
int i = (int) big;
System.out.printf("big = %f\n", big);
System.out.printf("i = %d\n", i);
```

Prints:
```
big = 4294967294.000000
i = 2147483647
```
Other Approaches to Coercion

Java differs from C in many ways, but one of the trickiest for inexperienced C programmers is that narrowing coercions by assignment are allowed.

```c
// valid C code
double num = 3.7;
int i = num / 4;
```

Another detail that is often missed: what happens on overflow?

```c
double num = 2147483647 + 1.0;
int i = num;
```

This is **undefined** behavior in C.

After execution on some compilers it is **possible** that \( i == 2147483647 \), but on others it can be **something else**!

Polymorphism

- An additional complication, in object-oriented languages, has to do with **subtype/supertype conformance**
  - In Java, a class type conforms to all **ancestors** (less specific supertypes) in its type hierarchy
  - A class does **not** conform to **descendants** (more specific subtypes) in its type hierarchy
- In Java, an identifier can be **polymorphic**: it can identify different specific types of objects at various times, so long as all of them conform to the type of the identifier itself

Java Conformance and Polymorphism

```java
class Vehicle {
    public String toString() {
        return "I'm a Vehicle!";
    }
}
class Car extends Vehicle {
    public String toString() {
        return "I'm a Car!";
    }
}
class Truck extends Vehicle {
    public String toString() {
        return "I'm a Truck!";
    }
}
```

Here, the Car and Truck types both conform to Vehicle
They **don’t** conform to each other
Vehicle **doesn’t** conform to Car or Truck, either

```java
public class Vehicle {
    public String toString() {
        return "I'm a Vehicle!";
    }
}
```

```java
public class Car extends Vehicle {
    public String toString() {
        return "I'm a Car!";
    }
}
```

```java
public class Truck extends Vehicle {
    public String toString() {
        return "I'm a Truck!";
    }
}
```

The vehicle variable is polymorphic.
The version of `toString()` that is called depends upon the actual object in memory at the time of the call.
Java Polymorphism and Type Checking

- While the specific types in play change **dynamically** at runtime, the Java compiler still does **static** type checks
- It doesn’t “wait to see what happens”
- It verifies at compile time that the types **will** conform
- The following doesn’t become legal, just because we have polymorphism and more complex type hierarchies

```java
Vehicle vehicle = new Vehicle();
System.out.println(vehicle);
vehicle = new Car();
System.out.println(vehicle);
vehicle = "I'm a Vehicle, too!";
System.out.println(vehicle);
```

Compile error: can’t convert String to Vehicle

---

This Week

- **Topic:** Types, Chapter 7, 8.1–8.4 of text
- **Schedule changes:**
  - Next two assignments pushed back one week
  - No class on Friday, 05 April (next week): MICS conference
- **Office Hours:** Wing 210
  - Monday: 9:00 AM – 10:30 AM
  - Tuesday: 3:00 PM – 4:00 PM
  - Wednesday: 9:00 AM – 10:30 AM
  - Thursday: 2:00 PM – 3:00 PM
  - Friday: 9:00 AM – 10:30 AM

---