Types in Java

• Types tell you the class of values from which a variable is drawn

• In Java we specify types for
  – Parameters
  – Variables
  – Return values
  – Class Fields

• Typically, we encode information described by elements of one or more different types
Types & Legal Operations

• For a given type, only certain “operators” can be used e.g.
  – e.g. a double precision value can be divided, multiplied, turned into a String etc.
  – A boolean can be tested for truthhood, turned into a String, etc.
  – A (reference to a) string can be used to
    • Extract prefixes or suffixes, find the length, concatenated, etc.
  – An enum’s values can be turned into a String, converted to integer, etc.
Java Primitive Types

- These are built-in to the Java language
- Primitive types in Java are the following
  - boolean
  - double
  - short (small integer)
  - int
  - char
  - byte
  - long
  - float
Non-Primitive Types

• Most types we used are not primitive types

• These are defined either
  – In our code
  – In the standard Java libraries
Why Types?

• Like specifying dimensions for an object (e.g. L, L³/T), specifying types lets us
  – Know what we’re dealing with (so we know what to do with it)
  – Avoid making a silly mistake
    • e.g. attempting to divide a number by a (reference to) a Person
    • Absent types, it is likely that these mistakes wouldn’t be identified until runtime
      – If we don’t happen to test that portion of the program, we won’t be aware of the error
  • With types, we can discover these errors when we are building the program -- during our “Build”
Type Coercion (“Casting”): Why

• Sometimes we have something that is a member of one type, but that can be logically converted to another type

• Examples:
  – We have a double-precision value and we wish to convert it instead to an integer (by dropping fractional component)
  – We have an integer (or a double, char, boolean, etc.) and wish to convert it to a string
  – (Subtyping) We have an ActiveObject that we know is a Person and wish to treat it as a Person
Type Coercion ("Casting"): How

• To "cast" a value $v$ in one type to another type, the following syntax is used:
  $(\text{TargetType}) \; v$

• Examples:
  
  traceln((String) age)
  
  ((Female) item).stateChart.isStateActive(Pregnant)
  
  ((int) age) + 1
Parameterized Types

- Sometimes a type (A) is defined in terms of another type (B)
  - This allows the definition of A to take & give back information specific to type B
    - e.g. methods take an A as a “parameter”, or return a B, etc.

- Common examples:
  - Collections dependent on type of their content (“set of double precision values”, “a dictionary mapping strings to integer”)
  - Tuples: “a pair of a character string and integer”

- We say that the type A is “parameterized by” type B
Parameterized Types

We can describe such “Parameterized Types” using Java “Generics”; Syntax used: A<B>, e.g. Set<Double>

– Here, the definition of one class can be defined with respect to an arbitrary number of classes that are provided via “Type parameters”

• Examples: ArrayList<ClassName>, Set<ClassName>
  
  This is an array list and set that can hold any type of classes (as specified by “ClassName”)

• A given use of such a “Generic” class will specify a specific class name for the type parameter e.g. Set<Person>, ArrayList<Double>, List<Deer>

• The definition of the generic can restrict the types that can be used for the type parameter via constraints
Examples of Parameterized Type (Generics)

• A resource pool depending on what resources are included (ResourcePool<MyResourceUnit>)

• An “array list” (like an extensible vector) depending on the type of the elements (ArrayList<Person>)

• Hypothetical: A Pair defined in terms of the first and second element
  – Pair<String, Double>
Examples of Type Parameterization in AnyLogic

• Experiment<MainClass> (and other experiment classes)
• ResourcePool<ResourceUnit>
• NetworkResourcePool<ResourceUnit>
• ActiveObjectArrayList<ActiveObject>
  Typically used (among other things) for the population in a main class
• ActiveObjectList<ActiveObject>
Example of a Parameterized Type
Enums: Why

- Often we desire in our models to encode categorical information
- We can certainly encode such information using integers (or shorts, etc.)
  - e.g.
    - Male=0, Female=1
    - Province: NL=0,NB=1,PEI=2,QC=3,etc.

- Example using variables
  ```
  int sex
  int province
  ```
Problem: This is fragile

– We could easily mistake a value “0” as encoding either Males or Newfoundland/Labrador

– e.g.
  • Reversing order of parameters given to a method, or entered into a file
  • Assigning value for one to another, due to a poorly named values

• e.g.
sex=province
Enumerations

- Enumerations help avoid manifest constants, group common names
- Good for bitwise operations: Consider values that will allow this rather than combinatorial names
- If language does not support enumerations, use carefully named global constants
- Leverage compiler checking
- If no class prefix, consider naming enumeration values with prefix giving type enumeration
- Make default enumeration value illegal
- Always explicitly handle all values
Enums in Java

• Enums let us
  – Give names to such information
  – Refer to the names in our code
  – Convert the names (where necessary) into their associated values
  – Compare names
  – Define operations on names
Simplest Examples

• enum Sex { Male, Female };
• enum Province { NL, NB, PEI, QC, ON, MB, SK, AB, BC};
• Variables using enum:
  Sex sex
  Province province
• Causes error: sex=province
Example of Enums in AnyLogic

```
public enum Sex { Male, Female }
public enum Ethnicity { FirstNations, Minto, EastAsian, SouthAsian, Caucasian }
```
A Closer Look

Additional class code:

```java
public enum Sex { Male, Female; }
public enum Ethnicity { FirstNations, Metis, EastAsian, SouthAsian, Caucasian; }
```
Use of Enums to Delineate Possible Parameter Values

- **color**: V
- **circlesize**: V
- **CircleSize**: F
- **isInitiallyInfected**: F
- **PregnancyStatus**: F
- **Susceptible**: F
- **Infective**: F
- **NonPregnant**: F
- **Pregnant**: F
- **Death**: F

Properties:

- **sex**: Sex.Female
- **appearanceTime**: V
- **InitialAge**: V
- **CurrentAge**: V
- **FinalizeDeath**: F
- **FertilityRateAgeEthnicity**: F
- **PerformBirth**: F
- **EstablishOffspringConnectionsBasedOnMothersConnections**: F
- **EstablishOffspringLocationBasedOnMothersLocation**: F
Use of Enums to Delineate Possible Parameter Values
Generating Random Possible Values

![Diagram showing processes and properties]

### RandomEthnicity - Function

<table>
<thead>
<tr>
<th>General</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name: RandomEthnicity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access: default</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Return Type:**
- [ ] void
- [ ] boolean
- [ ] int
- [ ] double
- [ ] String
- Other: Person.Ethnicity

**Function arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
</table>


// we pick a random one of the ethnicity values, by index (the value of the indices
int i = random.nextInt(Person.Ethnicity.values().length);
// now return the enum value associated with that index!
return Ethnicity.values()[i];