Type Checking

- Java compiler checks to see if your code is legal
- Today: Explore how this works
  - What is Java doing? Why
  - What will Java do if it sees a problem?
  - How can we help Java understand what we intended so that it can convert between object types?

The need for type checking

- Java programs use many types
  - Primitive types: int, char, long, double, ....
  - Predefined class types: Integer, String, Filestream, ...
  - Predefined interfaces
  - New types (classes or interfaces) that you might declare

Some easy casting situations

- Consider this code:
  ```java
double radius, circumference;
...
circumference = 2 * Math.PI * radius * radius;
```
  - Did a cast occur?
    - ... as it turns out, no. Java uses the type of constant (2 in this case) appropriate to the expression, so it treated 2 as 2.0
    - Java handles such things silently and automatically

- What about this:
  ```java
long ladderHeight;
int nSteps;
....
ladderHeight = nSteps * 11; // Assumes 11 inch/step
```
  - Question to think about:
    - Did Java start by computing nSteps*11 using 32-bit integer arithmetic, then convert to a long?
    - Or did Java convert nSteps to a long first? If so, it would interpret 11 as a long too...
    - The difference could matter: risk of an overflow

A cast can make code predictable

- If we write
  ```java
ladderHeight = (long)nSteps * 11;
```
  - we can be certain that long arithmetic is used
  - Note: The cast operator, a unary prefix operator, has priority over "*". Sometimes you end up having to add extra ()...
Java sometimes requires a cast

$$\text{long } x;$$
$$\text{int } a;$$

$$...$$
$$a = x; \quad \text{Type error; Java won’t cast long to int automatically}$$

- Java forces you to explicitly cast x to int, this way:
  $$a = (\text{int})x;$$

Why does Java have this rule?

This code doesn’t do anything illegal, but Java forces you to explicitly cast x to int, this way:

$$a = (\text{int})x;$$

... Java wants to be sure you realize that some longs won’t fit in an int. (64 versus 32 bits). Truncation occurs

Casting with objects

- To understand how casting works for objects, need to understand how Java determines an object’s type
  - Suppose MyLittlePony is a subclass of Toy and we write this code:
    $$\text{Toy myToy} = \text{new MyLittlePony(“SparkleDust”, …);}$$
  - What’s the type of myToy?
    $$\text{myToy MyLittlePony@x1}$$

Static versus dynamic typing

[Note: Unrelated to keyword static!]
- We’re given
  $$\text{Toy myToy = new MyLittlePony(“SparkleDust”, …);}$$
- myToy has “static” type Toy
- ... but “dynamic” or “instance” type MyLittlePony

Dynamic (“runtime”) types

- Dynamic type of a variable: type of the object assigned to it — it’s in the name of the object
  - myToy is a reference to an object of class MyLittlePony
  - Thus the dynamic type of myToy is MyLittlePony

Variable’s dynamic type may change at runtime whenever a new object is assigned to it.

Static (“compile time”) types

- The static type of a variable is the type with which it was declared.
  - In our example, myToy was declared to be of type Toy
  - Thus the static type of myToy is Toy
How does Java type check?

- Java needs to match the operators used in an expression to corresponding type-specific methods.
- Occurs in two steps:
  - First, expression must pass static type-checking analysis.
    - Ideally, this would guarantee type safety, but in practice there are some forms of errors that can't be sensed until runtime.
    - Example: casting x of type Object to Toy, is not legal if the dynamic type of the object is String.
  - At runtime, the dynamic type determines the actual methods used to perform the requested operations.

Examples of dynamic checks

- Consider:
  ```java
  Object x = something;
  ((Toy)x).PushTheButton(HowHard.Medium);
  ```
  - ... this code is legal if x is an object that either is a Toy or subclasses class Toy.
  - ... but is illegal for an object of type Lodging.

Casting

- You can treat an object as if it was anything in the type hierarchy above or below it.
- This include interface types: If an object implements an interface, you can treat it as having the type of that interface.
  - If Java senses a possible ambiguity it will force you to explicitly cast; otherwise it won't complain.
  - But static type checking is surprisingly hard and there are cases that "should" be checkable that aren't handled correctly just the same.

instanceof

- `ob instanceof C`:
  - `true` if `ob` is not null and object `ob` can be cast to class (or interface) `C`; `false` otherwise.
- Needed if `ob` is to be cast to `C` in order to use fields or methods declared in class `C` that are not overridden.

Example: Lodgings and Hotels

- Suppose we have a list of `Lodgings`.
  - It will be a "generic" type, a topic we'll explore soon.
  - The simplest example is a list: `List<T>`, as in:
  ```java
  List<RouteNode> route = findRoute("Ithaca", "Miami");
  List<Lodging> lodgingOptions = new List<Lodging>();
  for (RouteNode r: route)
    if (r instanceof Lodging)
      lodgingOptions.add( (Lodging)r );
  ```

... we end up with a list containing

- Hotels and motels
- Campgrounds
- Youth hostels
- ... anything that extends class `Lodging`
Method invocations

- When we call a method in an object, we always get the implementation defined by the dynamic type of the object, i.e. the overriding method, even if we are treating the object as an instance of another type!
- Strange case: the expression is type-checked using static types, but executed using dynamic types!
- To see this, consider:
  myToy.pushTheButton(HowHard.Lightly);
  ... Will compile only if Toy has method pushTheButton

Operators are really methods too

- In Java, operators (things like +, -, *, /, %, ....) are actually a shorthand for invoking methods
- Oddly, however, they don’t let you overload them
- C#, which grew out of Java, does allow this
- Operator overloading can be convenient, and many people complain that this is a mistake in Java
  - For example, in C# you can define a method to compare two Toys. Perhaps t1 < t2 if the store makes less money on t1 than on t2 or the store is trying to clear t2’s from inventory.
  - There is no obvious reason Java doesn’t allow this

... but now suppose that

- What if myToy was created using
  Toy myToy = new MyLittlePony(“FeatherFluff”, ...);
  Suppose MyLittlePony overrides method pushTheButton
  ... then myToy.pushTheButton(Lightly)
  calls the method in MyLittlePony, not the one defined in Toy!

Operators are really methods too

- In Java, operators (things like +, -, *, /, %, ....) are actually a shorthand for invoking methods
- Oddly, however, they don’t let you overload them
- C#, which grew out of Java, does allow this
- Operator overloading can be convenient, and many people complain that this is a mistake in Java
  - For example, in C# you can define a method to compare two Toys. Perhaps t1 < t2 if the store makes less money on t1 than on t2 or the store is trying to clear t2’s from inventory.
  - There is no obvious reason Java doesn’t allow this

... so?

- ... so this helps understand exactly why the dynamic definition of toString() is always the one that runs if you write code like
  System.out.println("Why not stay at " + place + "?");
- Moreover, this happens even if place is of type Lodging or RouteNode or even object.

... even so

- Everything we’ve said about methods also applies to operators
- For example, when you add an integer to a string:
  - String s = “Sparkle ate ” + howmany “ candies”;
  - Java starts by seeing String = String + int + String
  - Java autoboxes the int howmany is replaced by new Integer(howmany).
  - Then notices that string defines a + operator
  - Integer has a toString() method, so Java invokes it
  - Now we have string = (String + String) + String

Generics

- We briefly saw an example of a generic
  List<RouteNode> route = ... route.Add(new Hotel("Bates Hotel", ...));
  for (RouteNode r: route)
    if (r.youWillDieHere())
      System.out.println("Maybe we shouldn’t stop at " + r);
- A generic is just a type that takes other types as parameters, like a “list of RouteNodes”
Types and generics

- We'll get more fancy, but can already discuss a point that confuses some people
- Suppose method X is declared as:
  ```java
  public void X(List<Object> myList) { … }
  ```
- Should it be legal to invoke X(route)?
  - ... Java says no!

Seems like a cast should work

- What about X((List<Object>)route)?
  - Java still says no!
  - It claims route can’t be converted to a List<Object>
  - Why?
- ... to understand, think about what operations can be done on a List<Object>
  - Such as, for example,
    ```java
    myList.Add(new Integer(27));  // Add an object to myList
    ```

Not every object is a RouteNode!

- Example illustrates frustrating limit with strong types
  - In many situations, treating a List<RouteNode> object as a List<Object> would work perfectly well, like for sorting the list.
  - But Java won’t allow us to do that because some operations might fail at runtime and Java can’t tell if you plan to only do the “legal” kind!
  - Casting doesn’t help because the cast itself isn’t possible; casting is only possible if the two types are completely compatible

Generics deal with this

- The solution is to create entire classes in which types can be provided as parameters
- Then we can have a method to sort lists of type T
  - Like creating one version of the code for each type
  - Instead of sorting a List<Object>, it sorts List<T>
  - When you invoke it you specify T, e.g. RouteNode
- Very useful!

Types that support array indexing

- Java, like every language, has arrays
  ```java
  int[] myVector = new int[100];
  double[][] my2D = new double[8][32];
  ```
- ... and you can initialize them, of course
  ```java
  int[] myVector = new int[] { 9, 11, 2, 5 };
  ```
- Due to the magic of automatic type inference and a kind of operator overload, some other kinds of objects can also be treated just like arrays

But a 2D array isn’t what you expect

- ```java
  double[][] my2D = new double[10][20];
  double[][] myTriangle = new double[10][];
  for(int i = 0; i < 10; i++)
    myTriangle[i] = new double[i];
  my2D = myTriangle;
  ```
- What’s going on?
  - my2D was “really” a vector of 10 pointers... each capable of pointing to a vector of doubles
But a 2D array isn't what you expect

- This is in contrast to languages like MatLab and C# where you have “true” n-dimensional arrays
- Accessed as my2D[ i ][ j ], not my2D[ i, j ]

Array vs ArrayList vs HashMap (latter two from java.util)

- Array
  - Storage is allocated when array created; cannot change
  - Extremely fast lookups
- ArrayList (in java.util)
  - An “extensible” array
  - Can append or insert elements, access element i, reset to 0 length
  - Lookup is slower than an array
- HashMap (in java.util)
  - Save data indexed by keys
  - Can look up data by its key
  - Can get an iteration of the keys or values
  - Storage allocated as needed but works best if you can anticipate need and tell it at creation time.

What is an ArrayList?

- A pre-existing class in Java.util
  - ArrayList<String> myList = new ArrayList<String>();
  - myList.put("apples");
  - myList.put("pears");
  - System.out.println("Fruit 0 is " + myList.get(0));
- A list that can mimic an array
  - In Java arrays are fixed size, and this can be annoying (although you can “resize” them)
  - An ArrayList has variable size
  - The real underlying structure is a list of elements

ArrayList behavior

- Create: ArrayList<T> myAL = new ArrayList<T>();
  - T can be any object type you wish
  - But it can’t be a primitive type. So use Integer, not int, Boolean, not boolean, etc.
  - On the other hand, int[] would be legal because an array is an object
- An ArrayList behaves much like a normal array
  - myAL.get(i) is the i'th element, and will be of type T
  - But… you can “add to the end” via myAL.put(...);
  - It gets longer as you do put operations

Why can’t we use “array indexing”?

- In many languages, array-like classes allow indexing
  - It would be nice to write myAL[i] for example
  - C#, which extends Java, does allow this. The compiler simply translates this notation to a get or put call.
- But Java doesn’t support that, hence for ArrayList you need to explicitly call myAL.get(i) and myAL.put(i)
  - Put has two overloads
    - One puts something at the end: myAL.put(something)
    - The other puts it at location i: myAL.put(i, something)
How about a HashMap?

- Similar idea, but now the array index itself can be objects of any type you like
  - Similar to ArrayList, you access items using method calls
  - But you can think of these as mapping directly to array indexing even though that notation isn't permitted
- Designed to deal with applications that often need to look for something in a long list
- With an Array or an ArrayList we might need to search the whole list, item by item, looking at values

HashMap Example

- Create HashMap of numbers. Use names of numbers as keys:
  ```java
  Map<String, Integer> numbers = new HashMap<String, Integer>();
  numbers.put("one", new Integer(1));
  numbers.put("three", new Integer(3));
  ```
- Retrieve a number:
  ```java
  Integer n = numbers.get("two");
  ```
  - Returns null if HashMap doesn't contain key
  - Can use numbers.containsKey(key) to check this

What's going on here?

- First, we're seeing another generic:
  ```java
  numbers = new HashMap<String, Integer>();
  ```
  - In words, "A fast way to index with a string and pull out an associated integer"
- The previous slide actually used an interface type:
  ```java
  Map<String, Integer> numbers = new HashMap<String, Integer>();
  ```
  - Works because
    ```java
    HashMap<KT,VT> implements Map<KT,VT>
    ```

... and what is a HashMap?

- Idea: offer super-fast lookup
  - Take the key (the string)
  - Compute its "hash code"
    - Takes any object as an input
    - Outputs random-looking number computed from the object
  - HashMap allocates a big vector, initially empty, and uses the hash code to select an entry. Call the vector hashVec
    ```java
    hashVec[key.hashCode() % hashVec.length] = new KeyValue(key, value)
    ```
  - So: a "1-step" way to find key and value. (Collisions are handled automatically but no need to explain how this works)

... and what is a HashMap?

- Java is strongly typed, but a single object has many types
  - Its declared type and type Object
  - Its superclass types, and interface types it implements
- Many languages (including some very close to Java) extend this notion to "overloads" of operators like + or – or even array indexing
  - It involves implementing a special kind of interface
  - Java community has pressed for this in Java too, but as of now, it hasn't happened