Abstract Classes + Hashing

Hitesh Boinpally
Summer 2023

Agenda

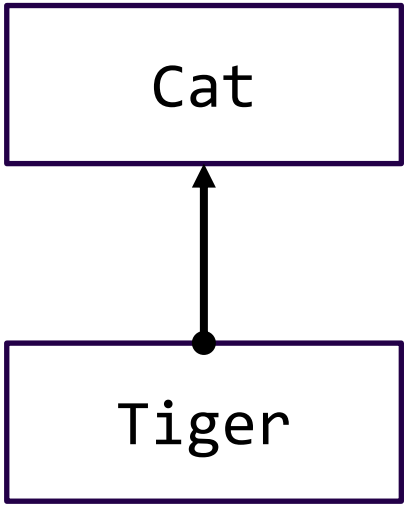
- Inheritance Review
- Abstract Classes
- Hashing

Inheritance

- **Inheritance:** Forming hierarchial relationships between classes
  - Allows for sharing / reusing of code between classes
  - **Superclass**: The class being extended
  - **Subclass:** The class that inherits behavior from superclass
    - Gains copy of every method

Inheritance

- **Inheritance:** Forming hierarchial relationships between classes
  - Allows for sharing / reusing of code between classes
  - **Superclass**: The class being extended
  - **Subclass:** The class that inherits behavior from superclass
    - Gains copy of every method
- Inheritance forms an “is-a” relationship
  - `Tiger extends Cat`
  - Means that `Tiger` “is-a” `Cat`



```
public class MusicPlayer {
    public void m1() {
        S.o.pln("MusicPlayer1");
    }
}
public class TapeDeck extends MusicPlayer {
    public void m3() {
        S.o.pln("TapeDeck3");
    }
}
```

```
public class IPod extends MusicPlayer {
    public void m2() {
        S.o.pln("IPod2");
        m1();
    }
}
public class IPhone extends IPod {
    public void m1() {
        S.o.pln("IPhone1");
        super.m1();
    }

    public void m3() {
        S.o.pln("IPhone3");
    }
}
```

	m1()	m2()	m3()
MusicPlayer			
TapeDeck			
IPod			
IPhone			

```
public class MusicPlayer {
    public void m1() {
        S.o.pln("MusicPlayer1");
    }
}
public class TapeDeck extends MusicPlayer {
    public void m3() {
        S.o.pln("TapeDeck3");
    }
}
```

```
public class IPod extends MusicPlayer {
    public void m2() {
        S.o.pln("IPod2");
        m1();
    }
}
public class IPhone extends IPod {
    public void m1() {
        S.o.pln("IPhone1");
        super.m1();
    }

    public void m3() {
        S.o.pln("IPhone3");
    }
}
```

	m1()	m2()	m3()
MusicPlayer	MP1	/	/
TapeDeck	MP1	/	TD3
IPod	MP1	IPod2 m1()	/
IPhone	IPhone1 MP1	IPod2 m1()	IPhone3

	m1()	m2()	m3()
MusicPlayer	MP1	/	/
TapeDeck	MP1	/	TD3
IPod	MP1	IPod2 m1()	/
IPhone	IPhone1 MP1	IPod2 m1()	IPhone3

```
MusicPlayer var1 = new TapeDeck();
MusicPlayer var2 = new IPod();
MusicPlayer var3 = new IPhone();
IPod var4 = new IPhone();
Object var5 = new IPod();
Object var6 = new MusicPlayer();


((TapeDeck) var1).m2();

((IPod) var3).m2();

((IPhone) var2).m1();

((TapeDeck) var3).m2();
```

	m1()	m2()	m3()
MusicPlayer	MP1	/	/
TapeDeck	MP1	/	TD3
IPod	MP1	IPod2 m1()	/
IPhone	IPhone1 MP1	IPod2 m1()	IPhone3

```
MusicPlayer var1 = new TapeDeck();
MusicPlayer var2 = new IPod();
MusicPlayer var3 = new IPhone();
IPod var4 = new IPhone();
Object var5 = new IPod();
Object var6 = new MusicPlayer();
```

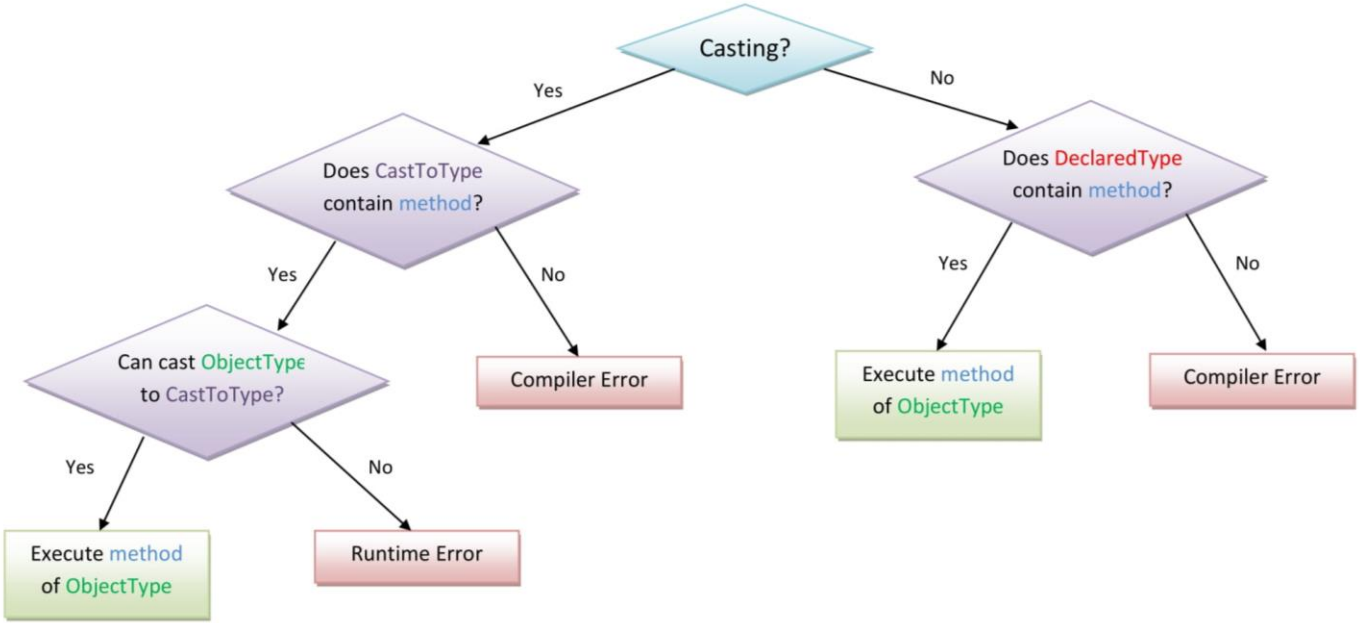
((TapeDeck) var1).m2();
Compiler Error (CE)
((IPod) var3).m2();
IPod2 / IPhone1 / MusicPlayer1
((IPhone) var2).m1();
Runtime Error (RE)
((TapeDeck) var3).m2();
Compiler Error (CE)

The Rules

First we define a few things with a color code

```
DeclaredType name = new ObjectType();    //declare variable
name.method();                           //call method
((CastToType)name).method();             //cast object, then call a method
```

When we try to execute one of the latter two, we follow this progression:



Abstract Classes

- Allow us to construct classes that leverage both inheritance and interface ideas
- Abstract classes cannot be instantiated (like interfaces)
- Include method implementations that can be leveraged with inheritance
- Can define abstract methods, which must be implemented by any subclass (like interfaces)

Agenda

- Inheritance Review
- Abstract Classes
- Hashing 

Recall: Arrays

- Allow for **random access** (continguous memory)
  - Have fast access if we know the index we are looking for
- Runtime of adding a value to an unsorted array?
- Runtime of checking if a value exists in an unsorted array?

Hashing

- **Idea**: Map every value for some object to some integer index
  - Store these values in an array based on the index (**hash table)**
- **Hash Function**: An algorithm to do this mapping
  - Idea for integers: HF`(x) = x % table.length`

Hashing

- **Idea**: Map every value for some object to some integer index
  - Store these values in an array based on the index (**hash table)**
- **Hash Function**: An algorithm to do this mapping
  - Idea for integers: `HF(x) = x % table.length`

```
set.add(11);                    // 11 % 10 == 1
set.add(49);                    // 49 % 10 == 9
set.add(24);                    // 24 % 10 == 4
set.add(7);                     //  7 % 10 == 7
```

index	0	1	2	3	4	5	6	7	8	9
value	0	**11**	0	0	**24**	0	0	**7**	0	**49**

Hashing Efficiency

```
public static int hashFunction(int i) {
    return Math.abs(i) % elementData.length;
}
```

- Add: set `elementData[HF(i)] = i;`
- Search: check `if elementData[HF(i)] == i`
- Remove: set `elementData[HF(i)] = 0;`

- What is the runtime of `add`, `contains`, and `remove`?
  - **O(1)!**
- Are there any problems with this approach?

“Good” Hash Functions

- Goal: Map an object to a number
- Requirements:
  - The same object should always have the same number
  - If two objects are considered “equal” they should have the same hash code

“Good” Hash Functions

- Goal: Map an object to a number
- Requirements:
  - The same object should always have the same number
  - If two objects are considered “equal” they should have the same hash code
- To be good:
  - Results should be distributed approximately uniformly
  - Should “look random”

“Good” Hash Functions

- Goal: Map an object to a number
- Requirements:
  - The same object should always have the same number
  - If two objects are considered “equal” they should have the same hash code
- To be good:
  - Results should be distributed approximately uniformly
  - Should “look random”

- How to write a hash function for `String` objects?

Hashing Objects

- The `hashCode` function inside `String` objects looks like this:

```
public int hashCode() {
    int hash = 0;
    for (int i = 0; i < this.length(); i++) {
        hash = 31 * hash + this.charAt(i);
    }
    return hash;
}
```

$$h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$$

- As with any general hashing function, collisions are possible.
  - Example: "Ea" and "FB" have the same hash value.
- Early versions of Java examined only the first 16 characters.
  For some common data this led to poor hash table performance.

Hashing Objects

- Hashing integers is easy (just mod by length)
- For objects, all Java objects contain the `hashCode` method (inherited from `Object` class)
  - `public int hashCode()`
  - Returns the hash code for an object

- `hashCode`'s implementation varies based on the object
  - You can define your own for your objects!

Hash function for objects

```
public static int hashFunction(E e) {
    return Math.abs(e.hashCode()) % elements.length;
}
```

- Add: set `elements[HF(o)] = o;`
- Search: check if `elements[HF(o)].equals(o)`
- Remove: set `elements[HF(o)] = null;`