Inheritance

- Inheritance is a way of extending functionality and properties of an existing class.
    - and allows you to add new features and overwrite existing ones.

Example

```
public class Person {

	private String name;
	private int age;

	public Person(String name, int age) {
		this.name = name;
		this.age = age;
	}

	public String getName() { return name; }
	public void setName(String name) { this.name = name; }
	public int getAge() { return age; }
	public void setAge(int age) { this.age = age; }
}
```

- Every person should have some identifiable name and age
- Many other classes can inherit this class for more specific representations of a Person.
    - Students, Soldier, Artist, Banker, Musician, Zombies, …

```
public class Student extends Person {

	private int studentID;

	public Student(String name, int age, int studentID) {
		// Need super() since we are extending from a Base Class
		// super implicitly gets called with Person default constr.
		super(name, age);
		this.studentID = studentID;
	}

	public int getStudentID() { return studentID; }

	public void setStudentID(int studentID) {
		this.studentID = studentID;
	}
}

//////
	public static void main(String[] args) {
		Student s = new Student("Richert", 21, 80498567);
		System.out.println(s.getAge()); // 21
		System.out.println(s.getName()); // Richert
		System.out.println(s.getStudentID()); // 80498567
	}
```

Some common lingo:

- We say that Person is the base class (or parent class) of Student
- Student is a derived class (or subclass) of Parent
- We can further derive subclasses from Student if we wished.
    - FullTimeStudent, PartTimeStudent, InternationalStudent, GraduateStudent, …
- Each of the subclasses that extend Student will inherit everything from Person AND Student as well as defining their own unique features within the class.

Example with class inheriting from Student

```
public class InternationalStudent extends Student {

	private double tuition;
	private String citizenship;

	public InternationalStudent(String name, int age, int id,
				double tuition, String citizenship) {
		super(name, age, id);
		this.tuition = tuition;
		this.citizenship = citizenship;
	}

	public double getTuition() { return tuition; }
	public void setTuition(double tuition) {
		this.tuition = tuition;
	}
	public String getCitizenship() { return citizenship; }
	public void setCitizenship(String citizenship) {
		this.citizenship = citizenship;
	}
}
```

Constructor Chaining

- InternationalStudent uses its super constructor and calls Student's constructor
- Student uses its super constructor and calls Person's constructor
- … and so on.
    - This is known as constructor chaining

Instanceof operator

- We can check if an object reference is a legal type using **instanceof** operator.
- This includes the current class and all base classes as well.
- instanceof works only for objects in the same hierarchy.
- Recall Exceptions
    - All Exceptions are of type Exception since they are all inheriting from that class
    - That's why we can catch ALL exceptions by catching an Exception type
- In order to actually check the specific class, you can use .getClass().equals()

Example

```
Student s = new Student("Richert", 21, 80498567);

if (s instanceof Person) {
      System.out.println("instance of Person"); // valid
}
if (s instanceof Student) {
      System.out.println("instance of Student"); // valid
}
if (s instanceof InternationalStudent) {
      System.out.println("instance of InterationalStudent"); // invalid
}
if (s instanceof Object) {
      System.out.println("instance of Object"); // valid
}
```

- In order to actually check the specific class, you can use .getClass().equals()

```
if (s.getClass().equals(Person.class)) {
      System.out.println("getClass.equals.Person");
}

if (s.getClass().equals(Student.class)) {
      System.out.println("getClass.equals.Student");
}

if (s.getClass().equals(Object.class)) {
      System.out.println("getClass.equals.Object");
}
```

Polymorphism (Dynamic Binding)

- Recall, all classes in Java extends the Object class.
- Which means that any object can be assigned to the Object class

```
Object o = new Student("Richert", 10,10);
Object o2 = new int[100];
Object o3 = new Scanner(System.in);
```

- But we can't have it the other way around
  - `Student s = new Object(); // illegal!`
- Writing code this way doesn't let you do much
  - Notice that only the Object methods are available…
  - This includes .toString(), .equals()…
    - .toString() defined in the object class is some sort of representation of the class itself
    - In most cases, this is probably not what you want .toString() to do.
    - .equals() in the Object class pretty much compares object references such as "== "
      - but recall that String.equals(String) compares the content.

How come String .equals() works differently if all objects inherit from Object.class?

- The String class provides its own implementation of how .equals() should work.
- The String class *overrides* the .equals() method and uses its own definition.
- A subclass can override inherited methods to provide its own specific definition.

Example

// Using our existing person / student classes

```
Person p = new Person("Richert", 10);
System.out.println(p.toString()); // Person@6d06d69c
```

- Unfortunately, this doesn't make much sense.
- and it may make sense for .toString() to provide some information on the state of the Class.
- We can override the .toString() method in our Person class:

```
public String toString() {
      System.out.println("In Person.toString()");
      return "Name: " + name + ", " + "Age: " + age;
}
```

Output:

```
In Person.toString()
Name: Richert, Age: 33
```

- Java recognizes that Person is the constructed object.
  - we said new Person(), not new Object().
  - Java will use the constructed type's methods if it exists
    - If not, it continues to look up the chain of inherited methods until it finds a match.
    - If it doesn't find a match, a compiler error occurs.
- The feature of matching the type of Object instantiated with the appropriate method definition is called ***dynamic binding*** or ***polymporphism***.

Final methods and classes

- We can declare methods and classes as **final.**
  - Declaring a class as final prevents other classes from extending it.
  - Declaring a method as final prevents other classes from overriding it.

Example

- Change .toString() in Person to: public final String toString() {
  - See the compiler error in Student and International Student saying it cannot override a final method.

- Change the class definition in Person to: public final class Person {
    - See the compiler error in Student stating that it cannot inherit from Person.

A polymorphism Example

```
Person[] personArray = new Person[3];
personArray[0] = new Person("A", 10);
personArray[1] = new Student("B", 1, 65);
personArray[2] = new InternationalStudent("C", 2, 5, 7.1, "US");

for (int i = 0; i < 3; i++) {
      // call .toString() for each type of object (polymorphism)
      System.out.println(personArray[i].toString());
}
```

Memory Slicing

- Primitive types

```
double x = 2.2;
int y = (int) x; // type casted
```

- Technically, primitive types are stored on the stack
- Memory slicing does occur in this case (i.e. only can represent the data within the size of the int).

- Objects

```
Object o = new Person("Richert", 21);
System.out.println(o.toString());
```

- Technically, Person is created on the "heap" in Java
- All Objects are created on the heap using "new"
- This includes all memory associated with type that was constructed
- o only is able to call methods that are defined in that class
- .toString() in Person is used due to **polymorphism**

Casting Objects

- Several rules about this: http://www.wideskills.com/java-tutorial/java-object-typecasting

```
Person p = (Person) new Object(); // runtime error
Person p = (Person) new Student("Richert", 21, 80498567); // OK
Object o = new Person("Richert", 21); // OK
Person p = (Person) o; // OK
// As long as you're casting compatible objects, then that's OK.
```

Abstract Methods

- Sometimes it doesn't make sense for a class to provide method definitions
- For example
  - A shape's area is dependent on the type of shape.
  - … or an Animal class may make a sound, but that sound is dependent on the Animal.
  - … or a Student may pay different tuitions (international students, out-of-state, international)… or the Unit limit may differ…
- If it doesn't make sense for the class to provide a method definition, but know that the method should be implemented somewhere in the subclasses.
  - Abstract methods can be used!
  - **A method can be declared as "abstract" if there is no sensible solution to provide an implementation in the base class**

Example

- Add "abstract" to Student class
- Add two abstract methods: calculateQuarterlyFees(), getUnitLimit()

```
public abstract class Student extends Person {

	private int studentID;

	public Student(String name, int age, int studentID) {
		super(name, age);
		this.studentID = studentID;
	}

	public int getStudentID() { return studentID; }

	public void setStudentID(int studentID) {
		this.studentID = studentID;
	}

	public String toString() {
		System.out.println("In Student.toString()");
		return super.toString() + ", studentID: " + studentID;
	}

	public abstract double calculateQuarterlyFees();
	public abstract int getUnitLimit();
}
```

- Subclasses that extend an abstract class have two choices:
  1. Fill in the "hole" by overriding and providing an implementation for the abstract methods in the base class.
  2. Declare itself as abstract and force subclasses to implement its abstract methods.
     - It's not necessary to redefine the abstract method, but it's good style so people who need to extend the abstract class won't "dig" through the hierarchy.

Example

// add this to InternationalStudent.java

```
public double calculateQuarterlyFees() {
      return tuition / 3;
}

public int getUnitLimit() {
      return 18;
}
```

// PartTimeStudent.java

```
public class PartTimeStudent extends Student {

      private int enrolledUnits;

      public PartTimeStudent(String name, int age, int studentID,
                             int units) {
            super(name, age, studentID);
            enrolledUnits = units;
      }

      public double calculateQuarterlyFees() {
            return 400 * enrolledUnits;
      }

      public int getUnitLimit() {
            return 11;
      }
}
```

Another polymorphism example using abstract methods.

```
Student[] array = new Student[2];
array[0] = new InternationalStudent("Rich", 21, 80498567, 6000, "US");
array[1] = new PartTimeStudent("A", 22, 5555555, 8);

for (int i = 0; i < 2; i++) {
      System.out.println(array[i].getName());
      System.out.println(array[i].calculateQuarterlyFees());
      System.out.println(array[i].getUnitLimit());
}
```

Interfaces

- In Java, a class may not extend multiple classes (unlike C++).
  - Though multiple inheritance is "kinda" possible.
- An **interface** is a mechanism for defining a "purely abstract class"
  - Interfaces may contain only public non-static methods and public static final fields (i.e. constants).
  - It's used when you want to specify a behavior that a class (and subclasses) need to support without specifying any implementation details.
  - A class may **implement** multiple Interfaces.

Example

```
public interface Comparable
{
      // compares two objects: x.compareTo(y) such that
      // if x == y, return 0
      // if x < y, return -1
      // if x > y, return 1
    int compareTo(Object obj);
}
```

- For each method in an interface, you have to provide a contract explaining what the meaning of the method means.
    - What does the returned int mean when compareTo is called?
    - You have to specify this in writing…

Example

```
public abstract class Student extends Person implements Comparable {
//…

// Implement the Interface's method in the Student class
// We can compare students based on their StudentID
// compares studentIDs
public int compareTo(Object o) {
      Student x = (Student) o;
      if (studentID < x.studentID) {
            return -1;
      } else if (studentID > x.studentID) {
            return 1;
      } else {
            return 0;
      }
}
```

// in main

```
Student[] array = new Student[2];
array[0] = new InternationalStudent("Richert", 21,80498567,6000, "US");
array[1] = new PartTimeStudent("A", 22, 5555555, 8);
System.out.println(array[0].compareTo(array[1])); // returns 1
System.out.println(findMinStudent().toString());

/////
public static Student findMinStudent() {
      Comparable[] c = new Comparable[2];
      c[0]= new InternationalStudent("Richert",21,80498567,6000, "US");
      c[1]= new PartTimeStudent("A", 22, 5555555, 8);
      Student minStudent = (Student) c[0];
      for (int i = 1; i < c.length; i++) {
            if (c[i].compareTo(minStudent) < 0) {
                  minStudent = (Student) c[i];
            }
      }
      return minStudent;
}
```